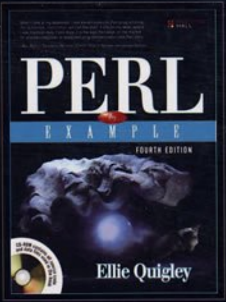
PRENTICE HALL

华章程序员书库

PERL 实例精解（原书第4版）

PERL by Example, 4E

（美） Ellie Quigley 著
陈宗斌 等译



经典的Perl学习指南
数万Perl程序员的案头必备

机械工业出版社
China Machine Press

第 1 章　实用摘要和报表语言

1.1　什么是 Perl

"懒惰、性急和傲慢。伟大的 Perl 程序员拥有这些优点。"

——Larry Wall

Perl 是一种多用途的开源（免费软件）解释型语言，由称为 Perl Porters 的核心开发团队维护和改进。它主要用作脚本语言，并且运行在众多平台上。尽管 Perl 最初是为 UNIX 操作系统设计的，但是它以其可移植性以及现在与大多数操作系统捆绑在一起而著名。这些操作系统包括 RedHat Linux、Solaris、FreeBSD、Macintosh 等。由于 Perl 的通用性，它通常称为程序设计语言的"瑞士军刀"。

Larry Wall 编写 Perl 语言来管理散布在网络中的日志文件和报表。依据 Wikipedia.org 上面的说法："Perl 最初命名为'Pearl'，它出自于 Gospel of Matthew（《圣经马太福音》）中的'Parable of the Pearl'（珍珠的寓言）。"该寓言的简要内容如下：一位商人寻找珍珠，他找到一颗如此贵重、漂亮的珍珠，以至于他情愿倾其所有来购买它。最后，他甚至比以往更富有。无论你怎样解释这则寓言，它都具有非常积极的寓意。

但是在 1987 年推出其官方发布版时，删去了"Pearl"中的字母"a"，自此将该语言称为"Perl"，后来又把它称为实用摘要和报表语言（Practical Extraction and Report Language），还有一些人把它称为病态折衷垃圾列表器（Pathologically Eclectic Rubbish Lister）。你很快将会看到，Perl 确实远远胜过实用报表语言或折衷垃圾列表器。Perl 使编程简单、灵活和快速，因此使用它的人都会喜欢它。其用户从经验丰富的程序员一直到只具有很少计算机知识的初学者，而且用户的数量也在飞速增长。[⊖]

Perl 传承自 UNIX。Perl 脚本在功能上类似于 UNIX awk、sed、shell 脚本和 C 程序。shell 脚本主要由 UNIX 命令构成，Perl 脚本则不然。sed 和 awk 用于编辑和报告文件，但是 Perl 无需执行文件即可工作。C 没有 shell、sed 和 awk 的任何模式匹配和通配元字符，而 Perl 却有扩展字符集。Perl 最初用于操作文件中的文本、从文件中提取数据和编写报表，但经过不断的发展，它现在可以操作进程、执行网络任务、处理 Web 页面、与数据库通信，以及分析科学数据。Perl 确实是程序设计语言的"瑞士军刀"，任何人都可以使用它。

⊖ 当指语言时，将 Perl 拼写为"Perl"；当指解释器时，则拼写为"Perl"。

本书中的示例是在 Solaris、Linux、Macintosh UNIX 和 Win32 系统上创建的。

Perl 通常与 O'Reilly Media 的商标即骆驼标志相关联，O'Reilly Media 出版了第一本关于 Perl 的书籍，书名是《Programming Perl》，作者是 Larry Wall 和 Randal Schwartz，该书称为“Camel Book”（骆驼书）。

1.2 什么是解释语言

在编写 Perl 程序时，读者需要准备两样工具：一个文本编辑器和一个 Perl 解释器。读者可以从许多 Web 站点上下载到后者，譬如 Perl.org、cpan.org 以及 activestate.com。和 C++、Java 等编译语言不同，读者无需在执行程序之前将其编译成机器能理解的代码。Perl 解释器会代劳这一切，它能完成程序的编译、解释和执行工作。像 Perl 这样的解释语言有很多优点，首先，它能运行在几乎任何平台上；其次，它相对易于学习；最后，它的速度很快，并且灵活度很高。

像 Python、Java 和 Perl 等解释语言都用到了中间代码，以便将编译和解释这两个过程联系起来。它首先会把用户提供的代码编译成一种内部压缩格式，称之为字节码（bytecode）或连接代码（threaded code），然后交给解释器执行。在运行 Perl 程序时，读者会观察到两个阶段：编译阶段和执行阶段，后一阶段才会生成程序结果。如果程序中存在语法错误，譬如关键字拼写错误或缺了引号，编译器就会报错。即使通过了编译，程序在开始执行时也可能出现其他问题。成功通过上述两个阶段后，才可以做其他事情，譬如改进程序或者提升程序性能等。

此外，解释器还提供了许多命令行开关项（选项）以控制其行为模式，譬如语法检查、发送警告信息、遍历文件、执行语句、打开调试器等。本书后面章节将详细介绍这些开关项。

1.3 Perl 的用户

由于 Perl 内建函数能方便地处理进程和文件，再加上它是可移植的（即能运行在多种不同的平台上），因此 Perl 深受那些需要同时监控多种不同系统的系统管理员的欢迎。万维网（World Wide Web）的迅速发展大大提高了人们对 Perl 的兴趣。它现在已经成为最流行的 CGI 脚本编写语言，负责动态生成 Web 页面。即使现在出现了像 ASP.NET 之类的其他专用 Web 页面处理语言，Perl 还是一如既往地受到系统管理员、数据库管理员、科学家、遗传学家以及所有需要从文件里收集和处理数据的用户的欢迎。

任何人都可以使用 Perl。不过倘若读者已经熟悉 UNIX Shell 脚本、C 语言或其他从 C 派生的语言（例如 C++ 和 Java）的话，学起 Perl 来会更容易一些。对上述用户而言，转到 Perl 语言的难度相对较小。而对于那些缺乏编程经验的读者，则需要多花些时间来学习。不过只要学会了 Perl，您可能就没有必要去使用其他语言了。

如果读者熟悉像 awk、grep、sed 和 tr 这样的 UNIX 工具，就知道它们的语法其实各不相同，其选项和参数处理方式也不同，甚至不同工具所遵循的规则都互不相同。如果读者是一位 Shell 程序员，往往需要经历一连串艰苦的学习过程，学会使用各种应用工具、shell 元字符、正则表达式元字

符和不计其数的引用。此外，Shell 程序功能有限，速度还很慢。用户若要执行更复杂的算术任务、进程间通信任务或二进制数据处理任务，则不得不使用更高级的编程语言，譬如 C、C++ 或 Java。了解 C 的读者都应当知道，要想用它在文件里搜寻模式，或者与操作系统交互处理文件或命令的话，那可不是什么简单的事情。

Perl 集中了 Shell 程序、C 语言以及 UNIX 中 awk、grep、sed 和 tr 工具的优秀特性。由于它速度很快，且不受特定容量数据块的限制，因此很多系统管理员和数据库管理员都从传统的 Shell 脚本转向使用 Perl。C++ 和 Java 程序员还可以利用 Perl 5 中新增的面向对象特性，譬如创建可重复使用的可扩展模块。Perl 现在可以由其他语言生成，而其他语言也可嵌入到 Perl 中。所有使用 Perl 的用户在做每件事时都应当牢记这样一句话："道路往往不止一条"（http://www.oreilly.com/catalog/opensources/book/larry.html）。

在开始编写脚本时，读者并不需要了解有关 Perl 的所有内容，甚至可以不是一名程序员。本书将为读者学习 Perl 打下良好的基础，帮助读者发现其诸多功能和优点。然后读者便可自行决定需要深入多少。如果不出意外，大家一定会发现 Perl 的乐趣所在。

1.3.1 Perl 的版本

Perl 有很多版本，其中两个主要版本是 Perl 4 和 Perl 5。Perl 4 的最后一个版本是 Perl 4.036，发布于很久之前的 1992 年。Perl 5.000 同样显得很古老，它发布于 1994 年秋天，其源代码经过彻底重写，对语言进行了优化，并引入了对象以及很多其他特性。尽管变化巨大，Perl 5 仍然保持着与以前版本的兼容性。本书示例都在这两个版本下进行了测试，若有区别之处，本书都将予以指明。在编写本书时，Perl 的最新版本是 Perl 5.8.8。下一代的 Perl 6 将再次对 Perl 进行重新设计，目前距离其官方发布还遥遥无期。尽管 Perl 6 可能会引入新的特性，但读者从本书学到的基本语法将永远保持不变。

1.3.2 什么是 Perl 6

> "Perl 5 是我本人对 Perl 进行的改写。我希望 Perl 6 将由全社区完成改写，并能属于整个社区。"
>
> —— Larry Wall，State of the Onion Speech，TPC4

Perl 6 在本质上与 Perl 5 类似，所不同的是它加入了许多新的特性，但其基本语法、特性和目标都将维持不变。如果读者对 Perl 已经有所了解，那么这些知识都不用推倒重来。如果按照本书内容学习 Perl，读者便能为 Perl 6 的正式发布做好准备。学习 Perl 6 的过程就好像是从美式英语转向澳大利亚英语，而不像从英语转向汉语那样麻烦。

如需获取有关 Perl 6 进展情况的进一步消息，请读者访问：

http://www.perl.com/pub/a/2006/01/12/what_is_perl_6.html?page=2

此外，如需了解 Larry Wall 生平和 Perl 的历史，请访问：

http://www.softpanorama.org/People/Wall/index.shtml#Perl_history

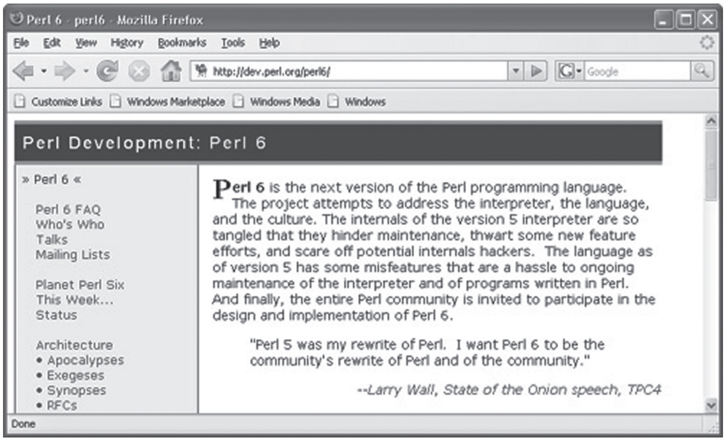


图 1-1 Perl 6 开发 Web 页面

1.4 如何获得 Perl

读者可以通过多种途径获得 Perl 发布包，其中最常用的是 Perl 综合文献网（Comprehensive Perl Archive Network，CPAN）。（www.cpan.org。）

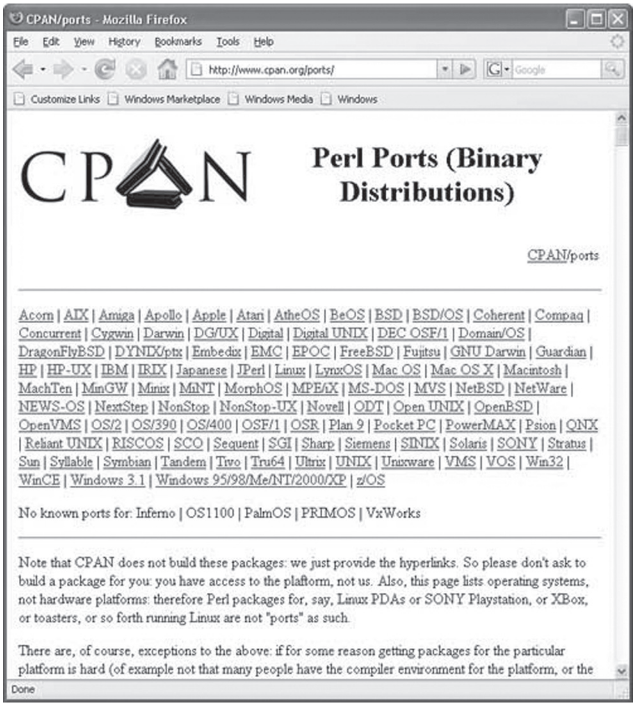


图 1-2 CPAN 中的二进制包发布页面

如需了解自己的系统平台可以运行哪种 Perl 发布包，可访问 http://www.cpan.org/port。读者

若想方便快速地完成 Perl 的安装，则可以使用 ActivePerl 组件。它是基于标准 Perl 源代码构建的一种完整的自安装包，支持 Windows、Max OS X、Linux、Solaris、AIX 和 HP-UX 等平台。它由 ActiveState 网站（www.activestate.com）在线发布。完整的 ActivePerl 安装包包含一份二进制形式的 Perl 核心包，以及所有相关在线文档。

下面几个 Web 站点将有助于读者了解更多有关 Perl 的信息。

- 由 O’Reilly Media 公司负责维护的 Perl 官方主页：www.Perl.com。
- 由 Perl 基金会运营的 Perl 目录网站（Perl Directory），该网站的目标是成为囊括一切 Perl 相

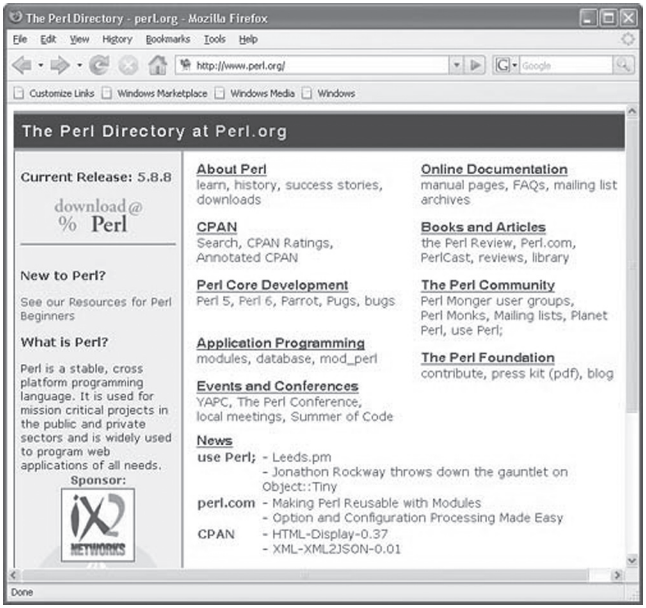


图 1-3　含有资源链接的 Perl 目录页面

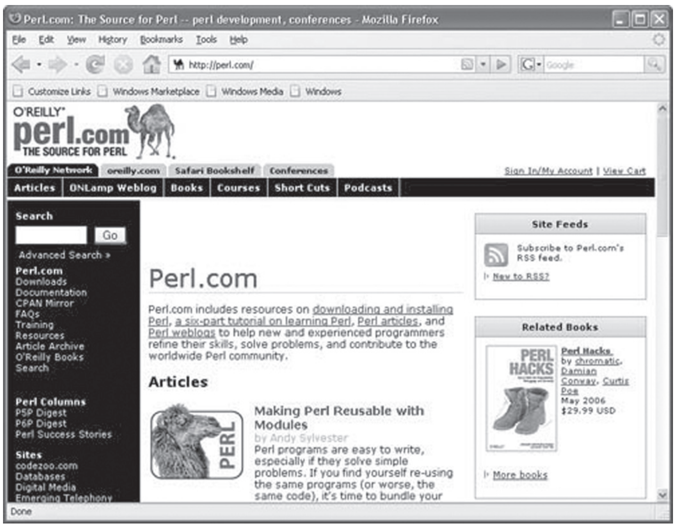


图 1-4　Perl 官方主页（O’ Reilly Media 公司负责运营）

关信息的中央目录：www.perl.org。

- Perl 综合文献网，在这里读者也能找到有关 Perl 的所有信息：http://www.cpan.org。
- 读者可以在这个网站里找到有关 Perl 开发的常用工具：http://www.activestate.com

Perl 的可用版本

如需获得有关 Perl 版本、当前二进制版本日期、补丁或其他版权信息的话，请输入下列命令，参见示例 1.1 所示（其中 $ 符号是 Shell 的提示符）。

示例 1.1

```
$ perl -v
1   This is perl, v5.8.8 built for MSWin32-x86-multi-thread
    (with 50 registered patches, see perl -V for more detail)

2   Copyright 1987-2006, Larry Wall

3   Binary build 820 [274739] provided by ActiveState
    http://www.ActiveState.com
    Built Jan 23 2007 15:57:46
4   Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
    the Perl 5 source kit.
        Complete documentation for Perl, including FAQ lists, should be
    found on this system using "man perl" or "perldoc perl".  If you
    have access to the Internet, point your browser at
    http://www.perl.org/, the Perl Home Page.

    This is perl, v5.8.8 built for MSWin32-x86-multi-thread
    (with 1 registered patch, see perl -V for more detail)

5   Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
    the Perl 5.0 source kit.
      Complete documentation for Perl, including FAQ lists, should be
    found on this system using man perl or perldoc perl. If you have
    access to the Internet, point your browser to www.perl.com/, the
    Perl home page.

    --------------------------------------------------------------

6   perl -v
    This is perl, v5.8.3 built for sun4-solaris-thread-multi
    (with 8 registered patches, see perl -V for more detail)

    Copyright 1987-2003, Larry Wall

    Binary build 809 provided by ActiveState Corp.
    http://www.ActiveState.com
    ActiveState is a division of Sophos.
    Built Feb  3 2004 00:32:12

    Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
```

```
        the Perl 5 source kit.

        Complete documentation for Perl, including FAQ lists, should be
        found on this system using 'man perl' or 'perldoc perl'.  If you
        have access to the Internet, point your browser at
        http://www.perl.com/, the Perl Home Page.
```

解释：

1. 示例所用 Perl 的版本是 5.8.8，是由 ActiveState 提供的 Windows 版。
2. Perl 的开发者 Larry Wall 拥有上述代码的版权。
3. 上述内容来自 ActiveState。
4. 读者可在遵守 Artistic Licence 或 GNU 相关条款的前提下复制 Perl。Perl 的发布过程遵循自由软件基金会（Free Software Foundation）制定的 GNU 许可证，也就是说，Perl 是免费的。
5. 对 Solaris（UNIX）而言，Perl 的最新版本是 5.8.3。

1.5 什么是 CPAN

CPAN是通向所有Perl相关知识的大门，其全称是Perl综合文献网（Comprehensive Perl Archive Network）。该 Web 站点拥有读者所需要的所有免费 Perl 资料，包括文档、FAQ、模块和脚本、二进制发布包与源代码、以及相关通知等。全世界有多个 CPAN 的镜像站点，读者可从如下 URL 获得距离最近的镜像：

www.perl.com/CPAN

www.cpan.org

如果读者想要为手头工作寻找某个 Perl 模块的话，不妨可以去 CPAN 看看。CPAN 的搜索引擎能在大量不同的分类中搜索所需模块。有关模块的知识将在本书第 12 章中予以介绍。

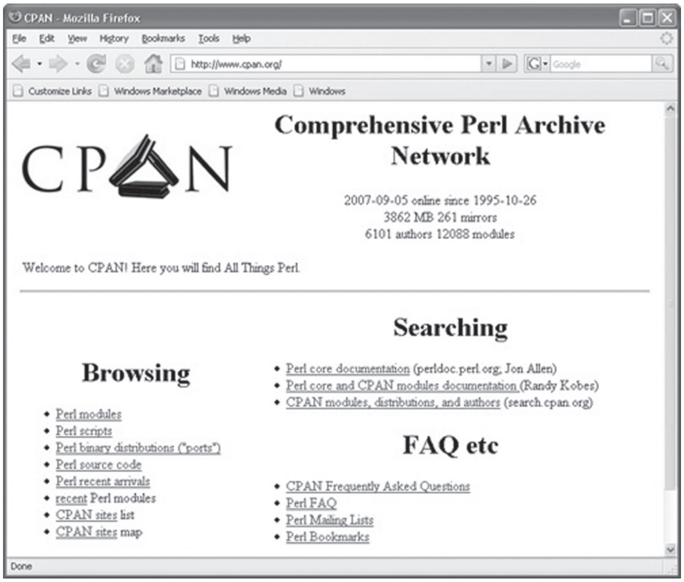


图 1-5 一个内容全面的 Perl 模块索引

1.6 Perl 文档

1.6.1 Perl 的 man 页面

标准的 Perl 发布包中包含了一份完整的在线文档，称为 man 页面，能为所有标准的 Perl 实用工具提供帮助信息（这个名字来自于 UNIX 系统中的 man）[⊖]。Perl 的 man 页面被划分为许多类别。如果读者在命令行提示符下输入下列内容：

```
man perl
```

就会得到所有类别的列表信息。因此，如果读者想要获得有关如何使用 Perl 正则表达式的帮助信息，可以输入：

```
man perlre
```

如果还需要获得有关子例程的帮助，则可键入：

```
man perlsub
```

下面列出了所有 Perl 的类别，其中后面的部分只适用于在线参考手册：

perlbot	面向对象的技巧和示例
perldebug	调试
perldiag	诊断信息
perldsc	数据结构：介绍信息
perlform	格式
perlfunc	内建函数
perlipc	进程间通信
perllol	数据结构：有关列表（list）的列表
perlmod	模块
perlobj	对象
perlop	运算符和优先级顺序
perlpod	无格式的旧文档
perlre	正则表达式
perlref	引用
perlsock	套接字支持扩展
perlstyle	样式向导
perlsub	子例程
perltie	隐藏于简单变量后的对象
perltrap	捕获错误
perlvar	预定义变量

如需了解具体库模块的用途，读者可使用 perldoc 命令查询相关文档。例如，如果要了解 CGI.pm 模块，可在命令行输入：

⊖ man 页面的意思是用户手册 manual 页面。——译者注

```
perldoc CGI
```

就能显示有关 CGI.pm 模块的文档。如果键入：

```
perldoc English
```

则会显示有关 English.pm 模块的帮助文档。

如需获得某个特定 Perl 函数的文档，读者可以键入 perldoc -f 和函数名。例如，为了获得有关 localtime 函数的帮助信息，可在命令行提示符下面执行如下命令（执行前可能需要设置相应的 UNIX/DOS 路径）。

```
perldoc -f localtime
localtime EXPR
localtime
        Converts a time as returned by the time function to a 9-element
        list with the time analyzed for the local time zone. Typically
        used as follows:
           #   0    1    2     3     4    5     6     7     8
            ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime(time);
<continues>
```

1.6.2 HTML 文档

当您从 ActivePerl 网站（即从 ActiveState.com）下载 Perl 的时候，会发现该网站提供了出色的帮助文档。由图 1-6 可见，从这里可以链接到有关 Perl 的几乎每一项内容。

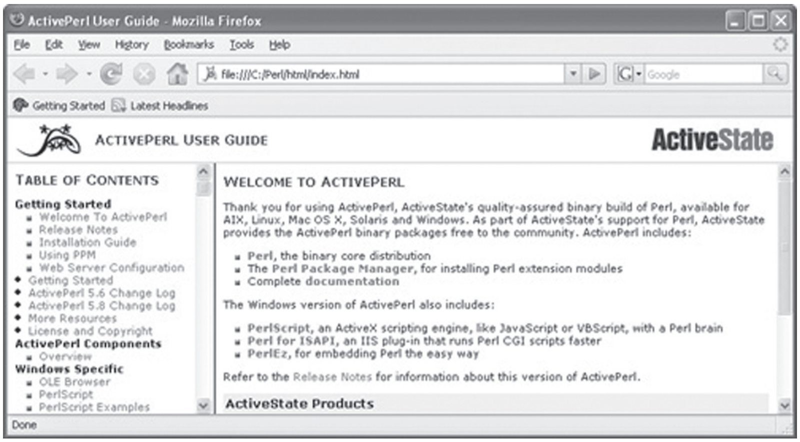


图 1-6 ActiveState 提供的 HTML Perl 文档

1.7 读者应当学到的知识

1. 谁编写了 Perl ？
2. Perl 这个词代表什么？
3. “开源” 的意思是什么？
4. Perl 的当前版本是多少？

5. 使用 Perl 的目标是什么?
6. 什么是解释器?
7. 从哪里能得到 Perl ?
8. 什么是 ActivePerl ?
9. 什么是 CPAN ?
10. 从哪里能得到帮助文档?
11. 如何获得特定 Perl 函数的帮助文档?

1.8 下章简介

在下一章，读者将学习如何创建一个简单的 Perl 脚本并执行它，以及 Perl 脚本的内部结构，包括 Perl 的语法、语句和注释。此外，读者还将学习如何检查语法错误，以及如何在命令行环境里使用多个选项执行 Perl 程序。

第 2 章　Perl 快速入门

2.1　快速入门和速查手册

2.1.1　给程序员的提示

如果读者已经拥有其他语言上的编程经验，如 Visual Basic、C/C++、Java、ASP 或 PHP 等；并熟悉基本的编程概念如变量、循环、条件语句、函数的话，表 2.1 将让您一览 Perl 语言的结构和语法。

在每个部分的结尾处，都会为读者提供该语法结构的相应章节号，并附有一段简短扼要的 Perl 程序示例，以显示如何使用该结构。

2.1.2　给非程序员的提示

如果您对编程一无所知，请跳过本章，直接阅读第 5 章。本章内容可作为读者以后学习过程中的速查手册。

2.1.3　Perl 语法和结构

表 2-1　Perl 语法和结构

脚本文件	Perl 脚本可通过文本编辑器来创建。一般情况下，用户无需为脚本提供特别的文件扩展名，除非执行该脚本的应用程序要求提供。譬如，如果是作为 Apache 容器中的 cgi 程序来执行脚本的话，就必须为脚本文件名提供 .pl 或 .cgi 扩展名
自由格式	Perl 是一种格式自由的语言。一个 Perl 语句必须由一个分号结尾，但它可以出现在程序的任意位置，亦可拆分为多行内容
注释	Perl 中的注释由 # 开头。解释器会在执行程序时自动忽略注释。注释可以出现在任何位置，但不能拆分为多行内容 **EXAMPLE** `print "Hello, world";  # This is a comment` `    #  And this is a comment`

（续）

<table>
<tr><td>打印输出</td><td>print 和 printf 是 Perl 的两个内建函数，用于显示输出内容。print 函数的参数是一系列由逗号隔开的字符串或数字。printf 函数则类似于 C 语言中的 printf() 函数，它能够归整输出内容的格式。用户无需在其参数两边提供括号（参阅第 3 章）
<pre>print value, value , value ;
printf (string format [, mixed args [, mixed ...]]);</pre>
<pre>EXAMPLE
print "Hello, world\n";
print "Hello,", " world\n";
print ("It's such a perfect day!\n"); #Parens optional;.
print "The the date and time are: ", localtime, "\n";
printf "Hello, world\n";
printf("Meet %s%:Age 5d%:Salary \$10.2f\n", "John", 40, 55000);</pre>
（参阅第 4 章）</td></tr>
<tr><td>数据类型 / 变量</td><td>Perl 变量支持三种基本的数据类型：标量、数组和关联数组（即散列 Hash）
Perl 变量无需在使用前声明
变量名以一个特殊字符（funny letter）开头，后面跟有任意数目的字母，包括下划线。其中，特殊字符表明该变量的类型和上下文语境。位于特殊字符后面的字母都是大小写敏感的。如果变量是以字母开头的话，其后便可跟随任意数目的字母（下划线等同于一个字母）和 / 或数字；如果变量不是以字母开头的话，则后面只能跟有一个字母（参阅第 5 章）</td></tr>
<tr><td>标量</td><td>标量是一种变量，只能保存单个值、单个字符串或单个数字。标量名必须以美元符号“$”开头。标量上下文则负责表明当前正在使用单个值的情况
<pre>EXAMPLE
$first_name = "Melanie";
$last_name = "Quigley";
$salary = 125000.00;
print $first_name, $last_name, $salary;</pre></td></tr>
<tr><td>数组</td><td>数组是一组有序排列的标量，如字符串和 / 或数字。数组中的各个元素由从 0 开始的整数来索引。数组变量名以“@”符号开头
<pre>@names = ("Jessica", "Michelle", "Linda");
print "$names"; #Prints the array with elements separated by a space
print "$names[0] and $names[2]"; #Prints "Jessica" and "Linda"
print "$names[-1]\n"; # Prints "Linda"
$names[3]="Nicole";   # Assign a new value as the 4th element</pre>
下面是一些常用的内建函数：
pop　移除最后一个元素
push　把新元素添加到数组末尾
shift　移除第一个元素
unshift　把新元素添加到数组开头
splice　在数组指定位置添加或移除数组元素
sort　对数组元素进行排序</td></tr>
<tr><td>散列</td><td>关联数组（associative array），又称为散列（hash），是一组未经排序的键 / 值对（key-value pair），并通过字符串进行索引。散列变量名以“%”号开头（请注意，若位于单引号或双引号中的话，%符号将不会解析）</td></tr>
</table>

（续）

散列

```
EXAMPLE
%employee =  (
   "Name"      => "Jessica Savage",
   "Phone"     => "(925) 555-1274",
   "Position"  => "CEO"
);

print "$employee{"Name"};  # Print a value
$employee{"SSN"}="999-333-2345";  # Assign a key/value
```

下面是一些常用的内建函数：

keys	检索散列数组中的所有键
values	检索散列数组中的所有值
each	检索散列数组中的某个键 / 值对
delete	删除某个键 / 值对

预定义变量

Perl 提供了大量的预定义变量。下面列举了常用的一些预定义变量：

$_	在执行输入和模式搜索操作时使用的默认空格变量
$.	文件中最后处理的当前行号
$@	由最近一个 eval() 运算符提供的 Perl 语法报错信息
$!	获取当前错误信息值，常用于 die 命令
$0	含有正在执行的程序名
$$	正在执行本脚本的 Perl 进程号
$PERL_VERSION / $^V	Perl 解释器的版本、子版本和修订版本信息
@ARGV	含有命令行参数
ARGV	一个特殊的文件句柄，用于遍历 @ ARGV 中出现的所有文件名
@INC	库文件的搜索路径
@_	在子例程中，@_ 变量含有传给该子例程的变量内容
%ENV	关联数组型变量 %ENV 含有当前环境信息
%SIG	关联数组型变量 %SIG 含有指向信号内容的句柄

常量（文本值）

固定不变的值，一旦设置就不能再更改。典型的常量包括 PI，或一英里长度的英尺数。这些值都是从不变化的。用户可借助 constant 保留字来定义常量，这里给出示例如下：

```
EXAMPLE
use constant BUFFER_SIZE => 4096;
use constant PI => 4 * atan2 1, 1;
use constant DEBUGGING => 0;
use contstant ISBN => "0-13-028251-0";
PI=6;  # Cannot modify PI; produces an error.
```

数字

Perl 支持整数（十进制、八进制以及十六进制整数）、浮点数、科学计数法、布尔型（Boolean）以及 null（空值）

```
EXAMPLE
$year = 2006;  # integer
$mode = 0775;  # octal number in base 8
$product_price = 29.95;  # floating point number in base 10
$favorite_color = 0x33CC99;  # integer in base 16 (hexadecimal)
$distance_to_moon=3.844e+5;  # floating point in scientific notation
$bits = 0b10110110;  # binary number
```

（续）

字符串和引号

所谓字符串，是位于引号内的一组字节（字符）。

如果要用引号来标识字符串的话，必须保证这些引号是成对出现的；譬如“String”或者‘String’。在双引号中，标量、数组变量（$x、@name）和反斜杠序列（如 \n、\t 和 \”等）都是可解释的。反斜杠可用于标识引号本身。单引号可嵌入到一对双引号之间，双引号也可嵌入到一对单引号之间。所谓 here 文档（here document），则是指一段嵌入于用户自定义标签中的文本块，其中第一个标签必须以 << 开头。

下面显示了引用一个字符串的三种不同方式：

```
单引号：'It rains in Spain';
双引号："It rains in Spain";
Here 文档：
   print<<END
      It
      rains in
   END Spain
EXAMPLE
$question = 'He asked her if she wouldn\'t mind going to Spain';
       # Single quotes
$answer = 'She said: "No, but it rains in Spain."';  # Single quotes
$line = "\tHe said he wouldn't take her to Spain\n";
$temperature = "78";
print "It is currently $temperature degrees";
       # Prints: "It is currently 78 degrees.". Variables are
       # Interpreted when enclosed  in double quotes, but not
       # single quotes
```

引用替换字符

Perl 为引号结构提供了一种替换形式。其中，引用的字符串将通过非字母形式的字符予以分隔，或由成对出现的字符包围起来，譬如 ()、{ }、[ ]。

下面使用的结构包括：qq，q，qw，qx

```
EXAMPLE
print qq/Hello\n/;   # same as: print "Hello\n";
print q/He owes $5.00/, \n"; # same as: print 'He owes $5.00', "\n";
@states=qw( ME MT CA FL );  # same as ("ME", "MT", "CA","FL")
$today = qx(date);  # same as $today = 'date';
```

运算符

Perl 提供了多种运算符，但是其主要内容与 C/C++、Java 或 PHP 运算符类似。运算符包括如下几种（详见第 6 章）

赋值运算符	=、+=、-= 、*=、%= 、^= 、&=、|= 、.=
算术相等	==、!=、<=>
字符串相等	eq、ne、cmp
算术大小关系	>、>=、<、<=
字符串大小关系	gt、ge、lt、le
范围运算符	5 .. 10 # 范围是 5 至 10 之间，逐个递增
逻辑运算符	&&、and、||、or、XOR、xor、!
自动递增 / 递减	++、--
文件运算符	-r、-w、-x、-o、-e、-z、-s、-f、-d、-l，等等
位运算符	~、&、|、^、<<、>>
字符串连接	.
字符串复制	x
代数运算符	*、/、-、+、%
模式匹配	=~、!~

```
EXAMPLE
print "\nArithmetic Operators\n";
print ((3+2) * (5-3)/2);

print "\nString Operators\n";  # Concatenation
print "\tTommy" . ' ' . "Savage";
print "\nComparison Operators\n";
print 5>=3 , "\n";
print 47==23 , "\n";
```

（续）

运算符

```
print "\nLogical Operators\n";
$a > $b && $b < 100
$answer eq "yes" || $money == 200

print "\nCombined Assignment Operators\n";
$a = 47;
$a += 3;          # short for $a = $a + 3
$a++;             # autoincrement
print $a;  # Prints 51

print "\nPattern Matching Operators\n"
$color = "green";
print if $color =~ /^gr/;     # $color matches a pattern
                              # starting with 'gr'
$answer = "Yes";
print if $answer !~ /[Yy]/;   # $answer matches a pattern
                              # containing 'Y' or 'y'
```

条件判断

基本的 if 语句能够判断小括号内的表达式取值，如果值为 true，则执行其后跟随的表达式。

if 语句

```
if ( 表达式 ) {
    语句;
}
```

```
EXAMPLE
if ( $a == $b ){ print "$a is equal to $b"; }
```

if/else 语句

if/else 实现了双向判断。如果 if 后面表达式条件为 true，则执行其后的语句 ；否则执行 else 后面的语句。

```
if ( 表达式 ) {
语句 ;
} else {
语句 ;
}
```

```
EXAMPLE
$coin_toss = int (rand(2 )) + 1; # Generate a random
                                 # number between 1 and 2
if( $coin_toss == 1 ) {
   print "You tossed HEAD\n";
}
else {
   print "You tossed TAIL\n";
}
```

if/elsif 语句

if/elsif/else 提供了多路分支选择功能。如果 if 后面的表达式值不是 true，则会依次判断其后每个 elsif 的条件，直到其中某个条件值为 true；如果没有一个为 true 的话，则执行最后的 else 语句。

```
if ( 表达式 ) {
    语句;
elsif（表达式）{
   语句;
} elsif（表达式）{
   语句;
else{
     语句;
}
```

（续）

条件判断

```
EXAMPLE
#  1 is Monday, 7 Sunday
$day_of_week = int(rand(7)) + 1;
print "Today is: $day_of_week\n";
if ( $day_of_week >=1 && $day_of_week <=4 ) {
   print "Business hours are from 9 am to 9 pm\n";
}
elsif ( $day_of_week == 5) {
   print "Business hours are from 9 am to 6 pm\n";
}
else {
   print "We are closed on weekends\n";
}
```

条件运算符

和 C/C++ 类似，Perl 也为 if/else 语法结构提供了相应的简化模式，该模式拥有两个运算符和三个操作数（因此又称为三元运算符）。如果条件值为 true，则执行紧跟在问号后面的语句；如果是 false，则执行冒号后面的语句。其格式为：

(条件) ? 条件为 true 时执行的语句 : 条件为 false 时执行的语句

```
EXAMPLE
$coin_toss = int (rand(2 )) + 1; # Generate a random number
                                 # between 1 and 2
print ($coin_toss == 1 ? "You tossed HEAD\n" : "You tossed TAIL\n" );
```

循环

循环结构能够指定一段重复执行多次的代码。Perl 支持多种不同类型的循环：while 循环、do-while 循环、for 循环以及 foreach 循环。

while/until 循环

while 循环：

while 后面跟随一个以小括号包围起来的表达式，以及一段执行语句。当表达式取值为 true 时，便继续执行循环内容。其语法结构为：

```
while（条件表达式）{
    代码段 A
}
EXAMPLE
$count=0;  # Initial value
while ($count < 10 ){  # Test
   print $n;
   $count++;  # Increment value
}
```

until 循环：

until 后面跟随一个以小括号包围起来的表达式，以及一段执行语句。当表达式取值为 false 时，继续执行循环内容。其语法结构为：

```
until（条件表达式）{
代码段 A
}
EXAMPLE
$count=0;  # Initial value
until ($count == 10 ){  # Test
   print $n;
   $count++;  # Increment value
}
```

do-while 循环

do-while 循环类似于 while 循环，所不同的是它是在循环体末尾检查循环表达式的，而不是在开头检查。这就保证了循环体至少能执行一次。

其代码结构为：

```
do{
代码段 A
}while（表达式）;
EXAMPLE
$count=0;  # Initial value
do {
   print "$n ";
   $count++;  # Increment value
   while ($count < 10 );  # Test
}
```

（续）

循环

for 循环

for 循环需要判断三个条件表达式的取值，各表达式之间以分号隔开。第一个表达式负责初始化变量，在整个循环过程中只调用一次。第二个表达式负责判断条件值是否为 true，如果是 true 的话便执行循环体；否则就退出循环。当循环体执行完毕后，控制权便转移到第三个表达式，由它负责更新待检测变量的值。然后，再由第二个表达式进行下一次判断，如此反复。其代码结构如下所示：

```
for(初始化鍬量;条件表达式;自加/自减){代嗎段
}
EXAMPLE
for( $count = 0; $count < 10; $count = $count + 1 ) {
   print "$count\n";
}
```

foreach 循环

foreach 循环仅用于逐个遍历列表（list）内容。

```
foreach$ 列表项（@ 列表）{
print$ 列表项, "\n";
}
EXAMPLE
@dessert =  ( "ice cream", "cake", "pudding", "fruit");

foreach $choice (@dessert){
    # Iterates through each element of the array
    echo "Dessert choice is: $choice\n";
}
```

循环控制

last 语句可用于从循环体中跳出一个循环。next 语句可用于跳过当前这次循环的剩余内容，直接从头开始下一轮循环。

```
EXAMPLE
$n=0;
while( $n < 10 ){
   print $n;
   if ($n == 3){
       last;  # Break out of loop
   }
   $n++;
}
print "Out of the loop.<br>";
EXAMPLE
for($n=0; $n<10; $n++){
   if ($n == 3){
       next;  # Start at top of loop;
              # skip remaining statements in block
   }
   echo "\$n = $n<br>";
}
print "Out of the loop.<br>";
```

子例程 / 函数

函数（function）是一组能完成某项任务的代码体，并供程序其他部分调用。用户可以通过参数将数据传送给这个函数。函数可以有返回值，也可以不返回任何值。任何合法的 Perl 代码均可出现在函数体的定义中。定义于函数外面的变量在函数体内也同样可用。而 my 函数则能把指定的变量局部化。

（参阅第 11 章）。

```
sub function_name{
   函数体
}
```

（续）

子例程 / 函数

```
EXAMPLE
sub greetings() {
   print "Welcome to Perl!<br>";  # Function definition
}
&greetings;  # Function call
greetings(); # Function call

EXAMPLE
$my_year = 2000;

if ( is_leap_year( $my_year ) ) {    # Call function with an argument
   print "$my_year is a leap year\n";
}
else {
   print "$my_year is not a leap year";
}

sub is_leap_year {    # Function definition

   my $year = shift(@_);  # Shift off the year from
                          # the parameter list, @_
   return (((($year % 4 == 0) && ($year % 100 != 0)) ||
   ($year % 400 == 0)) ? 1 : 0;    # What is returned from the function
}
```

文件处理

Perl 提供了 open 函数用于打开文件，也提供了 pipe，用于读写、追加文件内容。其中，open 函数的参数包括一个用户自定义的文件句柄（一般表现为一串大写字符），以及一个含有文件路径和读 / 写 / 追加标志的字符串（详见第 10 章）。

```
EXAMPLE
To open a file for reading:

open(FH, "<filename");      # Opens "filename" for reading.
                            # The < symbol is optional.
open (DB, "/home/ellie/myfile") or die "Can't open file: $!\n";

To open a file for writing:

open(FH, ">filename");     # Opens "filename" for writing.
                           # Creates or truncates file.

To open a file for appending:

open(FH, ">>filename");     # Opens "filename" for appending.
                            # Creates or appends to file.

To open a file for reading and writing:

open(FH, "+<filename");     # Opens "filename" for read, then write.
open(FH, "+>filename");     # Opens "filename" for write, then read.

To close a file:

close(FH);
```

（续）

文件处理

```
To read from a file:

while(<FH>){ print; }        # Read one line at a time from file.
@lines = <FH>;               # Slurp all lines into an array.
print "@lines\n";

To write to a file:

open(FH, ">file") or die "Can't open file: $!\n";

print FH "This line is written to the file just opened.\n";
print FH "And this line is also written to the file just opened.\n";
EXAMPLE

To Test File Attributes
print "File is readable, writeable, and executable\n" if -r $file and
-w _ and -x _;
            # Is it readble, writeable, and executable?
print "File was last modified ",-M $file, " days ago.\n";
            # When was it last modified?
print "File is a directory.\n " if -d $file;    # Is it a directory?
```

管道（Pipe）

管道用于把系统命令输出内容作为输入流传送给 Perl，或者将 Perl 的输出内容转发给系统命令以作为其输入。管道又称作过滤器（filter）。用户必须通过 open 系统调用来使用管道。该系统调用接受两个参数：一个用户自定义的句柄和一个操作系统命令，并需在操作系统命令的前面或者后面加上一个“|”符号。如果“|”符号出现在命令之前，则表示该命令将把 Perl 的输出作为输入内容；否则表示 Perl 将读取该命令的输出内容。也就是说，如果命令后面带有“|”，就表示 Perl 将从管道中读取输入内容；否则说明 Perl 将把输出内容写入到管道中去。

（详见第 10 章）

```
EXAMPLE
Input filter

open(F, " ls |") or die;    # Open a pipe to read from
while(<F>){ print ; }    # Prints list of UNIX files

Output filer
open(SORT, "| sort" ) or die;  # Open pipe to write to
print SORT "dogs\ncats\nbirds\n"
     #  Sorts birds, cats, dogs on separate lines.
```

正则表达式。所谓正则表达式，是由斜杠圈起的一组字符集合。它们可用于在文本中匹配指定模式，并进行相应替换操作。一直以来，Perl 都因为其优秀的模式匹配机制而闻名于世。（参阅第 8 章。）

表 2-2　一些正则表达式元字符

元字符	表达含义
^	匹配行首
$	匹配行尾
a.c	匹配一个 a，后面任意单个字符，再后面是一个 c 的情况

（续）

元字符	表达含义
[abc]	匹配 a 或者 b 或者 c 的情况
[^abc]	匹配字符既不是 a 又不是 b 也不是 c 的情况
[0-9]	匹配位于 0 到 9 之间的单个数字
ab*c	匹配一个 a 后面跟 0 到多个 b，最后是一个 c 的情况
ab+c	匹配一个 a 后面跟 1 到多个 b，最后是一个 c 的情况
ab?c	匹配一个 a 后面跟 0 到 1 个 b，最后是一个 c 的情况
(ab)+c	匹配 1 到多个 ab 后面跟着 1 个 c 的情况
(ab)(c)	捕获 ab 并将其值赋予变量 $1，同时捕获 c 值并赋予 $2

```
EXAMPLE
$_ = "looking for a needle in a haystack";
print if /needle/;
     If $_contains needle, the string is printed.

$_ = "looking for a needle in a haystack"; # Using regular expression metacharacters
print if  /^[Nn]..dle/;
    # characters and "dle".

$str = "I am feeling blue, blue, blue..."
$str =~ s/blue/upbeat/;  # Substitute first occurrence of "blue" with "upbeat"
print $str;
I am feeling upbeat, blue, blue...

$str="I am feeling BLue, BLUE...";
$str = ~ s/blue/upbeat/ig;  # Ignore case, global substitution
print $str;
I am feeling upbeat, upbeat...

$str = "Peace and War";
$str =~ s/(Peace) and (War)/$2, $1/i;  # $1 gets 'Peace', $2 gets' War'
print $str;
War and Peace.

$str = "He gave me 5 dollars.\n"
s/5/6*7/e; #  Rather than string substitution, evaluate replacement side
print $str;
He gave me 42 dollars."
```

在命令行下传送参数。Perl 通过 @ARGV 数组保存了命令行提供的参数内容。如果用到了 ARGV 文件句柄，则这些命令行参数将被视为文件；否则就视为是来自命令行环境的字符串，供脚本直接使用（详见第 10 章）。

```
EXAMPLE
$ perlscript filea fileb filec

(In Script)
```

```
print "@ARGV\n";  # lists arguments: filea fileb filec
while(<ARGV>){  # filehandle ARGV -- arguments treated as files
   print;  # Print each line of every file listed in @ARGV
}
```

引用，指针。 Perl 引用（reference）又称作是指针（pointer）。它在本质上是一个标量型变量，负责保存另一个变量的地址。在创建指针时，用户需要使用反斜杠运算符（详见第 13 章）。

```
EXAMPLE
# Create variables
$age = 25;
@siblings = qw("Nick", "Chet", "Susan","Dolly");
%home = ("owner" => "Bank of America",
         "price" => "negotiable",
         "style" => "Saltbox",
);

# Create pointer
$pointer1 = \$age;   # Create pointer to scalar
$pointer2 = \@siblings;  # Create pointer to array
$pointer3 = \%home; # Create pointer to hash
$pointer4 = [ qw(red yellow blue green) ]; # Create anonymous array
$pointer5 = { "Me" => "Maine", "Mt" => "Montana", "Fl" => "Florida" };
           # Create anonymous hash

# Dereference pointer
print $$pointer1; # Dereference pointer to scalar; prints: 25
print @$pointer2; # Dereference pointer to array;
        # prints: Nick Chet Susan Dolly
print %$pointer3; # Dereference pointer to hash;
        # prints: styleSaltboxpricenegotiableownerBank of America
print $pointer2->[1];   # prints "Chet"
print $pointer3->{"style"}; # prints "Saltbox"
print @{$pointer4}; # prints elements of anonymous array
```

对象。Perl 中的对象是一种特殊类型的变量。在 Perl 中，一个类代表着含有一组变量（属性）与函数（方法）的一个包。Perl 没有提供专门的 class 关键字。其中，属性就是能够描述这个对象的变量；而方法则是一种特殊的函数，允许用户创建并操作给定的对象。在创建对象时，用户需使用 bless 函数（详见第 14 章）。

创建一个类

```
EXAMPLE
package Pet

sub new{ # Constructor
    my $class = shift;
    my $pet = {
        "Name"  => undef,
        "Owner" => undef,
        "Type"  => undef,
    };
```

```
        bless($pet, $class);
    # Returns a pointer to the object

        sub set_pet{    # Accessor methods
            my $self = shift;
            my ($name, $owner, $type)= @_;
            $self->{'Name'} = $name;
            $self->{'Owner'}= $owner;
            $self->{'Type'}= $type;
        }
        sub get_pet{
        my $self = shift;
        while(($key,$value)=each($%self)){
            print "$key: $value\n";
        }
    }
```

实例化一个类

```
EXAMPLE
$cat = Pet->new();  # alternative form is: $cat = new Pet();
# Create an object with a constructor method
$cat->set_pet("Sneaky", "Mr. Jones", "Siamese");
# Access the object with an instance
$cat->get_pet;
```

此外，Perl 还支持方法的继承，为此必须把基类置于数组 @ISA 内。

库和模块。库文件都带有 .pl 扩展名；而模块则拥有 .pm 扩展名。目前，.pm 文件比 .pl 文件更为常用（详见第 12 章）。

库的路径

@INC 数组负责保存一组标准 Perl 库的路径列表。

导入外部文件

如需导入某个外部文件，应使用 require 或者 use 关键字。

```
require("getopts.pl"); # Loads library file at run time
use CGI;  # Loads CGI.pm module at compile time
```

诊断模式。如需在 Perl 脚本退出时获得出错原因等信息，用户可使用内建的 die 函数或者 exit 函数。

```
EXAMPLE
open(FH, "filename") or die "Couldn't open filename: $!\n";
if ($input !~ /^\d+$/){
       print STDERR "Bad input. Integer required.\n";
       exit(1);
}
```

此外，用户还可使用如下 Perl 选项：

```
use warnings;         # 提供警示信息，但不退出程序
use diagnostics;      # 提供详细的警示信息，但不退出程序
use strict;           # 检查全局变量、无引号单词等内容；允许退出程序
```

```
use Carp;                # 与 die 函数类似，但能提供更多程序出错信息
```

2.2 本章小结

本章是为那些希望快速浏览 Perl 全貌的程序员而准备的，介绍了 Perl 的一般语法与编程结构。本章仅给出了 Perl 的概况。本书后面的章节将帮助读者发掘更多有关 Perl 的知识。

在读者积累了一定编程经验之后，本章还可作为一个微型的实践教程，以帮助读者快速回忆起所需知识点，而不必抱着索引慢慢查找。

2.3 下章简介

在第 3 章中，我们会讨论 Perl 脚本的创建，譬如：如何命名一个脚本、如何执行脚本、以及如何添加注释、语句和内建函数。我们还将学习如何使用 Perl 的命令行开关，以及如何定位特定的一些错误。

第 3 章　Perl 脚本

3.1　创建脚本

下面是一个简单的 Perl 脚本：

示例 3.1

```
(The Script)
#!/usr/bin/perl
print "What is your name? ";
chomp($name = <STDIN>);  # Program waits for user input from keyboard
print "Welcome, $name, are you ready to learn Perl now? ";
chomp($response = <STDIN>);
$response=lc($response);  # response is converted to lowercase
if($response eq "yes" or $response eq "y"){
     print "Great! Let's get started learning Perl by example.\n";
}
else{
   print "O.K. Try again later.\n";
}
$now = localtime;  # Use a Perl function to get the date and time
print "$name, you ran this script on $now.\n";

(Output)
What is your name? Ellie
Welcome, Ellie, are you ready to learn Perl now? yes
Great! Let's get started learning Perl by example.
Ellie, you ran this script on Wed Apr  4 21:53:21 2007.
```

示例 3.1 提供了一个 Perl 脚本示例。在不远的将来，读者也能写出与之相似的脚本。Perl 脚本由一系列 Perl 语句和声明（declaration）组成。其中所有语句均以分号（;）结尾。（由于声明只在子例程和报表格式中才会用到，因此本书将在后面相关章节里予以介绍。）用户可以在脚本任意位置创建变量。如果变量没有初始化的话，系统将根据上下文语境自动把它赋值为 0 或 null。请读者留意，上述脚本中的变量名均以 $ 开头。用户可以把数字、文本字符串或函数输出值赋给变量。不同类型的变量通过其开头不同的特殊标志（funny symbol）予以区分，具体情况请参考第 4 章内容。

对每一条语句，Perl 将从头至尾仅执行一遍。

3.2 脚本

3.2.1 启动

UNIX/Mac 操作系统。如果脚本第一行含有以 #! 标志开头的 Perl 可执行文件全路径的话（又称为 shbang 行），就将告诉系统核心应当使用哪种程序来解释该脚本。下面是一个起始行的示例：

```
#!/usr/bin/perl
```

请务必注意，一定要在 shbang 行（即 #! 后面）中指定正确的解释器路径。不同系统中安装的 Perl 可能位于不同的目录下。在调用 Perl 编写的 CGI 脚本时，大多数 Web 服务器都会先读取这一行内容。因此，如果指定的路径与实际情况不符的话，将会引起严重错误。为了获知自己系统上 Perl 解释器的路径，读者可在 UNIX 提示符后键入[㊀]：

```
which perl
```

如果 shbang 行位于脚本的第一行，读者便可在命令行下通过其文件名直接执行该脚本。如果 shbang 行不在脚本的第一行，UNIX Shell 就会尝试把脚本解释为 shell 脚本，同时把 shbang 行当作注释来处理。（若要了解如何才能执行 Perl 脚本，请读者参阅“执行脚本”一节。）

Mac OS 在本质上属于 UNIX 的一个变种，因此它默认也带有 Perl 5.8。其使用方法和 Solaris、Linux、*BSD、HP-UX、AIX OS X 等操作系统完全相同。

Windows。Win32 平台并不支持 shbang 行或其他类似机制[㊁]。在 Windows XP 或者 Windows NT 4.0 [㊂]中，用户可以把 Perl 脚本关联到相应文件扩展名上，譬如 .pl 或者 .plx，然后便可直接从命令行执行脚本了。在命令行模式下或者系统控制面板中，用户可将 PATHEXT 环境变量设置成需要关联到 Perl 脚本上的文件扩展名。在命令行中，设置环境变量的命令是：

```
SET PATHEXT = .pl ; %PATHEXT%
```

在控制面板中，为了设置好上述关联信息，需执行如下操作：

1. 进入开始菜单。
2. 选择“设置”，或者直接选择“控制面板”。
3. 选择“控制面板”。
4. 在控制面板中，点击“系统”图标。
5. 点击“高级”。
6. 点击“环境变量”。
7. 点击“新建”。
8. 在变量名一栏中输入 PATHEXT。
9. 在变量值一栏中输入相应文件扩展名，并以分号和 %PATHEXT% 结尾。
10. 确定上述设置。

㊀ 另一种获取解释器路径的途径是使用：find / -name ‘*Perl*’ -print。

㊁ 尽管 Win32 平台一般不需要 shbang 行，但 Apache Web 服务器却需要它。因此如果读者编写的 CGI 脚本将由 Apache 负责执行的话，那还是必需提供 shbang 行。

㊂ 在 Windows 95 中，除非是以 Explorer 窗口打开应用程序，否则文件关联功能将无法使用。

现在，用户可以创建一个 Perl 脚本，并在其文件名中指定所选的扩展名，譬如 myscript.pl 或 myscript.plx。然后，只需在命令行中输入不带扩展名的脚本名称（如 myscript），便可直接执行该脚本了。（有关脚本执行方面的更多知识请参阅“执行脚本”一节。）

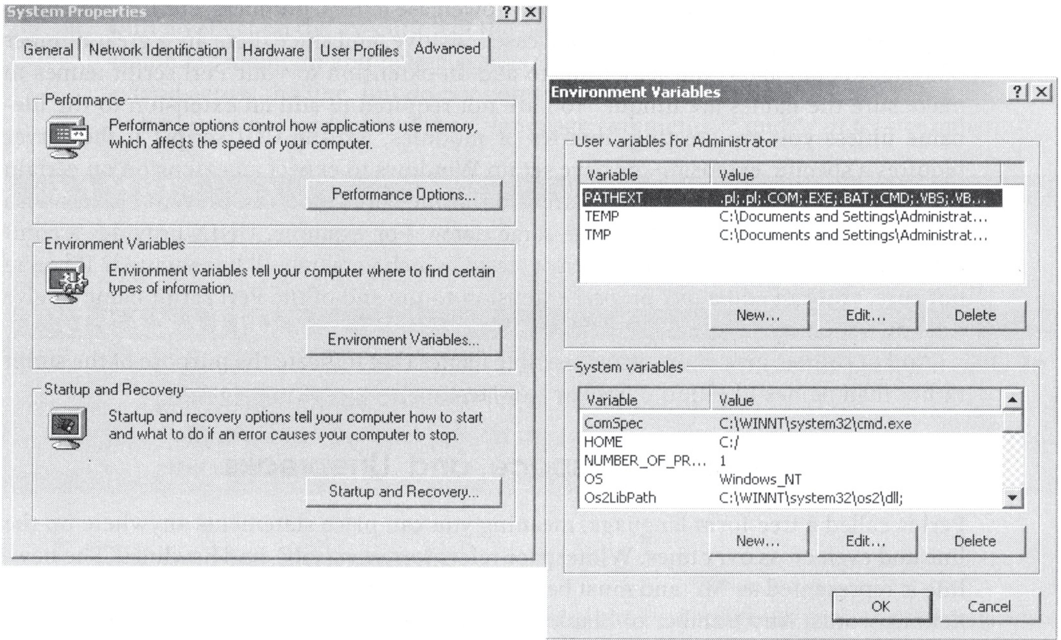


图 3-1　设置 PATHEXT 环境变量

3.2.2　选择文本编辑器

在编写 Perl 脚本时需要使用文本编辑器。为此，用户可以使用操作系统提供的任何编辑软件，也可从网上下载专门为 Perl 设计的更为高级的文本编辑器，包括各种第三方编辑器以及集成开发环境（Integrated Development Environment，即 IDE）。表 3-1 列举了常用的一些编辑器。

表 3-1　几种编辑器

BBEdit、JEdit	Macintosh
Wordpad、Notepad、UltraEdit、vim、PerlEdit、JEdit、TextPad	Windows
pico、vi、emacs、PerlEdit、JEdit	Linux/UNIX
Komodo	Linux、Mac OS、Windows
OptiPerl、PerlExpress	Windows
Affus	Mac OS X

3.2.3　为 Perl 脚本取名

在给 Perl 脚本取名时，惟一需要遵守的规则就是用户所在操作系统的文件命名规范（譬如大小

写字符、数字等等）。例如，如果当前操作系统是 Linux，文件名就是大小写敏感的；此外，由于存在着大量系统命令，因此最好给脚本加上一个扩展名，以便让它是全局惟一的。用户并不一定得为脚本提供扩展名，除非需要创建库或模块，或者服务器要求所写 CGI 脚本拥有指定的扩展名，又或 Windows 系统设置要求必须提供扩展名。一旦给脚本加上了惟一的扩展名，便可避免脚本和系统中其他程序的名字发生冲突。譬如，UNIX 系统已经提供了一个名叫“test”的命令。如果用户将某个脚本命名为“test”的话，那系统应当执行哪个“test”呢？如果没有把握的话，最好在 Perl 脚本文件名的末尾加上一个 .plx 或者 .Perl 扩展名，这样就能让脚本名全局惟一了。

当然，最好能为脚本准备具有实际含义的名称，以指明脚本的用途是什么。请读者不要使用像“foo”、“foobar”或“test”之类的名字。

3.2.4 语句、空白和换行

Perl 是一种格式自由的语言，意味着用户可以在任何位置放置脚本语句，甚至还能让同一个语句跨行出现。“空白”这个词包括空格、制表符以及换行符。Perl 脚本使用“\n”来表示换行符，并通过双引号将它包围起来。空白部分用于隔开不同的单词。在两个标记或单词之间可以出现任意多个空白字符。出现在引号内的空白部分将被保留；如果不在引号内，解释器则会忽略它们。下面这几个表达式表达了相同的意思。

5+4*2　　等价于　　5　+　4　*　2;

下面这两个 Perl 语句也都是正确的，其中用引号围起来的空白字符部分将会显示出来。

```
print "This is a Perl statement.";

        print "This
                    is
                      also
                        a Perl
                            statement.";
```

尽管用户在编写 Perl 脚本时拥有很多自由，但最好还是能将语句整理到各自所在行内，并为大段代码提供缩进（本书将在第 5 章讨论该内容）。当然，为自己的程序添加注释也是一样重要的，这样可以让自己和别人明确地理解各段程序分别在做什么。有关注释的更多知识请阅读下一节内容。

3.2.5 注释

用户可能当天写出了一段非常精彩的代码，但仅仅两周后就忘了这段代码是干什么用的。如果把这段代码发给别人，引起的困惑就更大了。所谓注释，其实是一段纯文本内容，允许用户在 Perl 脚本中插入说明性文本，同时对程序的执行不造成任何影响。其目的是帮助开发者或其他程序员对脚本进行维护和调试。Perl 注释都以 # 符号开头，其作用域一直能到本行末尾，但不延伸到下一行。

Perl 无法处理 C 语言提供的注释标识符 /* 和 */，或者 C++ 的注释标识符 //。

示例 3.2

```
# This is a comment
print "hello";  # And this is a comment
```

解释：

1. 正如 UNIX Shell、sed 和 awk 脚本一样，注释是以 # 号开头的代码行，并能延伸到这一行的末尾。
2. 注释可以出现在脚本的任意行中。上面这个示例显示的是一个注释后面跟随着一行合法的 Perl print 语句。

3.2.6 Perl 语句

Perl 脚本的大部分内容都是其可执行语句。与 C 语言一样，语句由一个或一系列表达式组成，并以分号结尾。Perl 语句分为简单语句和复合语句两种，由各种运算符、修饰符、表达式和函数组成，如下面示例所示。

```
print "Hello, to you!\n";
$now = localtime();
print "Today is $now.\n";
$result = 5 * 4 / 2;
print "Good-bye.\n";
```

3.2.7 使用 Perl 内建函数

对于任意一种编程语言来说，其内建的或打包于特殊库中的函数集都是该语言的重要组成部分（请参考附录 A.1）。Perl 也提供了很多有用的函数，这些函数都是独立的程序代码，负责完成某些特定的工作。在调用内建函数时，用户只需键入函数名，或者也可在函数名后带上一对小括号。所有的函数名都是小写的，其中许多函数还要求用户提供参数，也就是需要传给函数的消息。例如，如果用户不提供任何参数的话，print 函数就只能输出一行空白内容，因为其参数指定了需要在屏幕上输出的内容。如果函数需要参数的话，可将这些参数以逗号隔开，依次放在函数名后面即可。函数在完成特定任务后往往还会返回一些内容。在本章开头的示例中，调用了两个内建的 Perl 函数：print 与 localTime。其中，print 函数接受一个字符串参数，并将字符串的内容显示到屏幕上；而 localTime 函数则不需要任何参数，并能返回当前日期和时间。下面两个语句都是带着参数调用函数的合法形式。其参数都是“Hello，there.\n”。

```
print ( "Hello, there.\n " );
print  "Hello, there.\n ";
```

3.2.8 执行脚本

如果脚本中含有 #! 起始行并拥有执行权限（见示例 3.3），或者已经按前面所述步骤在 Windows 中设置好了文件扩展名关联，用户便可在命令行中直接通过脚本名称来执行 Perl 脚本。倘若 #! 不是脚本第一行的话，读者也可把脚本路径作为参数传给 Perl 解释器，这样便可执行该脚本。

接着，Perl 会通过其内部格式对用户提供的脚本进行编译和运行。如果脚本中存在语法错误，Perl 会马上告知用户。用户还可以通过 -c 开关检查脚本是否编译成功，如下所示：

```
$ perl  -c  scriptname
```

若要在 UNIX 或 MS-DOS 命令行提示符下执行脚本，请输入：

```
$ perl  scriptname
```

3.2.9 脚本实例

下面这段示例显示了 Perl 脚本的五个主要部分：

1. 起始行（UNIX）
2. 注释
3. 脚本体内的可执行语句
4. 检查 Perl 语法
5. 脚本的执行（UNIX、Windows）

示例 3.3

```
    $ cat first.perl       (UNIX display contents)
1   #!/usr/bin/perl
2   # My first Perl script

3   print "Hello to you and yours!\n";

4   $ perl -c first.perl   # The $ is the shell prompt
    first.perl syntax OK

5   $ chmod +x first.perl (UNIX)

6   $ first.perl or ./first.perl
7   Hello to you and yours!
```

解释：

1. 起始行，负责告诉 Shell 哪里有 Perl。
2. 注释部分，描述了程序员希望提供的有关该脚本的信息。
3. 可执行语句，其中含有一个 print 函数。
4. -c 开关，可用于检查语法错误。谢天谢地，上述程序一切正常。
5. chmod 命令，负责开启脚本的执行权限。
6. 执行脚本,（如果 UNIX 的默认路径中含有“.”目录的话）。如果显示“Command not found”错误（或其他类似反馈信息），请在脚本名前面加上一个句点和一条正斜杠。
7. 脚本会在屏幕上输出字符串“Hello to you and yours!”。

示例 3.4

```
    $ type first.perl      (MS-DOS display contents)
1   # No startup line; This is a comment.
2   # My first Perl script
3   print "Hello to you and yours!\n";
4   $ perl first.perl   (Both UNIX and Windows)
5   Hello to you and yours!
```

解释：

1. 上述示例中没有以 #! 开头的起始行。在使用 Windows 时，脚本并不一定需要起始行。如果读者使用的是 ActiveState，也可借助一个名叫 pl2bat 的实用工具创建脚本文件。

2. 这是一行描述性内容；注释内容说明了该脚本不含起始行。
3. 可执行语句，其中含有一个 print 函数。
4. 在命令行环境中，把脚本名称作为参数传给 Perl 程序，以便执行该脚本，并打印脚本输出内容。读者在任何操作系统上都可通过上述方式执行 Perl 脚本。

3.2.10　可能出现的错误

读者应当作好程序出错乃至大量出错的心理准备。为了得到一个能够完美运行的程序，读者可能需要再三尝试，屡败屡战。了解碰到的错误信息，和了解老板的怪癖、了解您的另一半、甚至和了解你自己一样重要。有些程序员就往往一而再、再而三地犯着同样的错误。不过也不用着急，读者马上就能学到大多数此类出错信息的含义，以及如何去避免它们。

在执行一段 Perl 脚本时，虽然读者经历的操作只有一步，但在 Perl 解释器内部却经历了两个阶段。首先，它会把整个程序编译成字节码，即该程序的一种内部表达。然后，由 Perl 的字节码引擎负责逐行运行这些字节码。如果碰到编译器错误，譬如行末缺少分号、关键字拼写错误、或引号不配对的话，读者便会看到所谓的语法出错信息。用户可通过 -c 开关收集此类错误信息，一旦熟悉之后，往往就能很容易地发现它们。

示例 3.5

```
    (The Script)
   print "Hello, world";
1  print "How are you doing?
2  print "Have you found any problems in this script?";

(Output)
Bareword found where operator expected at errors.plx line 3, near
"print "Have"
  (Might be a runaway multi-line "" string starting on line 2)
        (Do you need to predeclare print?)
syntax error at errors.plx line 3, near "print "Have you "
Search pattern not terminated at errors.plx line 3.
```

解释：

1. 这一行应当拥有配对的双引号，并在行尾添上一个分号。
2. 这个 Perl 语句本身是正确的。但是由于 Perl 正在搜寻上一行遗漏的双引号，因此会和本行的“print”一词产生混淆。这是因为 Perl 认为该行还是上一行的一部分。为什么呢？因为前一行缺少了一个引号，并且行末没有分号。因此，一旦读者在出错信息里面看到了“runaway”这个词，往往就意味着代码中某个引号已经 runaway（缺失）了。倘若看到“Bareword”一词，则意味着某个词语两边缺少引号。

当程序成功通过了编译阶段（也就是说，编译器没有发出任何语法错误或冲突消息）之后，用户还可能碰到一些运行时错误，又称作逻辑错误。这些错误相对更难以发现，可能是开发者没有预计到程序运行时才出现的某些问题所致，也有可能该程序在设计时的逻辑就是错的。运行时错误的原因很多，譬如打开的文件或数据库不存在、用户输入了错误的内容、陷入死循环、或者把 0 当作除数等。不论如何，这一类问题更难以排查，因而又被称为“Bug”。Perl 提供了一个调试器，允许用户逐行执行脚本中的语句，帮助排查逻辑错误的原因所在。（详见“调试器”一节。）

3.3 从命令行使用 Perl

虽然 Perl 的大部分功能都需要用到脚本，但对于那些简单的任务如测试函数、打印语句或只测试 Perl 语法，用户也可直接在命令行中予以执行。Perl 提供了大量命令行开关（switch），又称为命令行选项，用于控制或修改其行为表现。下列开关列表并不完整（完整的列表请参阅附录 A），但它基本说明了命令行下直接调用 Perl 的语法。

在使用命令行时，用户可以看到 shell 提示符。shell 又称为“命令行解释器”。其中，UNIX shell 如 Korn 和 bash 都默认显示 $ 提示符，而 C shell 则使用 % 提示符。UNIX、Linux 和 Mac OS 的 shell 在解析命令方面非常相似。在默认情况下，如果读者使用的是 WindowsXP 或 Vista 的话，会使用名叫 command.com 的 MS-DOS shell ；倘若使用的是 Windows NT，其 shell 则是 cmd.exe 中的控制台应用程序。上述二者均显示 $ 提示符[⊖]。Win32 shell 具有其独特的命令行分析方式。由于大多 Perl 编程工作都是在脚本文件中完成的，因此用户很少需要关心与 shell 的交互细节。但是，当把脚本与操作系统进行交互时，如果不知道该使用的命令以及 shell 如何执行它们的话，就有可能会发生问题。

3.3.1 –e 开关

–e 开关允许 Perl 从命令行而不是脚本来执行 Perl 语句。这是在把 Perl 语句放入脚本前检测其正确性的好办法。

示例 3.6

```
1   $ perl -e 'print "hello dolly\n";'      # UNIX/Linux
    hello dolly
2   $ perl -e "print qq/hello dolly\n/;"    # Windows and UNIX/Linux
    hello dolly
```

解释：

1. Perl 打印字符串 hello dolly 到屏幕，最后是一个换行符 \n。其中，美元符号（$）是 UNIX 的命令行提示符。把 Perl 语句包围起来的单引号负责在扫描和解释命令行时区分 Perl 命令和 UNIX shell。
2. 在 MS-DOS 提示符下，Perl 语句内容必须出现在双引号中间。其中，hello dolly 两侧出现的 qq 结构是 Perl 表示双引号的另一种形式。譬如，qq/hello/ 即等价于“hello”。如果在 MS-DOS 提示符下输入下列命令，则会显示出错信息：

   ```
   $ perl -e ' print "hello dolly\n" ; '
   Can't find string terminator " " anywhere before EOF at -e line 1.
   ```

请注意：UNIX 系统也可使用上述命令格式。

3.3.2 –n 开关

如果需要打印文件内容或搜索文件中含有特定模式的行，则可以使用 –n 开关隐式地逐一按行遍历文件。与 sed 和 awk 一样，Perl 也通过其强大的模式匹配技术在文本中查找目标模式。在使

⊖ 用户看到的命令行提示符也有可能被定制为含有当前目录路径、历史数、驱动盘符等内容。——译者注

用 −n 开关时，Perl 只会打印指定行的内容。

从文件读取。-n 开关允许用户遍历整个文件，只需在参数中提供该文件的文件名即可。其 Perl 语句位于引号之间，文件名则在命令行的尾部列出。

示例 3.7

```
(The Text File)
1   $ more emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75
2   $ perl -ne 'print;' emp.first      # Windows: use double quotes
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75

3   $ perl -ne 'print if /^Igor/;' emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
```

解释：

1. 在屏幕上打印文本文件 emp.first。在第 2 行，Perl 把这个文件名用作命令行参数。
2. 通过隐式地逐行遍历文件的方式，Perl 打印出 emp.first 文件的所有行。请注意，Windows 用户应当将命令放在双引号中，而非单引号。
3. Perl 通过正则表达式规定待匹配的模式。模式 Igot 出现在反斜杠之间，并在行首加上了一个 ^ 字符。^ 字符又被称为行首锚点（beginning of line anchor）。Perl 将只打印那些以 Igor 开头的行。请注意，Windows 用户应当将命令放在双引号中，而非单引号。

从管道读取。由于 Perl 只是又一个应用程序而已，因此用户也可通过管道把命令的输出内容传输给 Perl。Perl 将把来自管道而不是文件的内容当作输入内容。为此需要使用 -n 开关，以便让 Perl 遍历来自管道的所有内容。

示例 3.8

```
(UNIX)
1   $ date | perl -ne 'print "Today is $_";'
2   Today is Mon Mar 12 20:01:58 PDT 2007

(Windows)
3   $ date /T | perl -ne "print qq/Today is $_/;"
4   Today is Tue 04/24/2007
```

解释：

1. UNIX date 命令的输出内容通过管道传送到 Perl，存储在 $_ 变量中。然后，程序会在屏幕上打印引号之间的字符串 Today is 和 $_ 变量内容，并在最后追加一个换行符。
2. 输出 $_ 变量，其内容则是今天的日期值。
3. Windows NT 的 date 命令，它使用 /T 选项，能够生成当天日期。然后把输出内容通过管道传送给 Perl，并存储于 $_ 变量中。这里必须使用双引号括起 print 语句。

通过标准 I/O 的重定向操作，Perl 也可接受来自文件的输入，并能把输出重定向到文件中去。

示例 3.9

```
1   $ perl -ne 'print;' < emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75
2   $ perl -ne 'print' emp.first > emp.temp
```

解释：

1. Perl 的输入内容来自文件 emp.first，其输出内容则显示在终端屏幕上。请注意，Windows 用户应当将命令放在双引号中，而非单引号。
2. Perl 的输入内容来自文件 emp.first，其输出内容则被重定向到文件 emp.temp 中。请注意，Windows 用户应当将命令放在双引号中，而非单引号。

3.3.3 –c 开关

正如本章前面所述，–c 开关用于检查 Perl 语法，而不是执行 Perl 命令。如果其语法正确，Perl 就会通知用户。读者最好始终打开 –c 开关，以便时刻检查脚本的正确性。这对于 Perl 编写的 CGI 脚本显得尤为重要，因为这样便可把那些通常显示在屏幕上的出错信息发送到日志里去。

示例 3.10

```
1   print "hello';  Search pattern not terminated at  line 1.
 Can't find string terminator '"' anywhere before EOF at test.plx
2   print "hello";
    test.plx syntax OK
```

解释：

1. 字符串 hello 以双引号开头，却以单引号结束。引号必须是配对的，因此上面这个双引号应当和字符串末尾的另一个双引号配对。但实际上其末尾并没有双引号，出现的反而是单引号。通过打开 –c 开关，Perl 便会显示其在编译时是否发现了语法错误。
2. 纠正上述问题后，Perl 便会显示该语法是正确的。

3.4 读者应当学到的知识

1. 如何创建一个脚本？
2. 每个语句是以什么符号结尾的？
3. 什么是空白字符？
4. 格式自由（free form）的涵义是什么？
5. 什么是内建函数
6. 什么是 UNIX 中的 #! 行？
7. 如何设置脚本使它变得可执行？

8. 为什么要使用注释？
9. 如果一个 Perl 脚本没有 shbang 行，应当如何执行它？
10. 哪个命令行选项提供对 Perl 语法的检查？
11. −e 开关的作用是什么？

3.5 下章简介

如果不能在程序中打印出其所完成的工作内容，那就和尝试理解哑巴的思想一样麻烦。在下一章内容中，我们将讨论用于向屏幕（stdout）打印输出内容的 Perl 函数，以及如何归整输出的格式。读者将了解 Perl 是如何看待字符、空格、文本、反斜杠序列、数字和字符串的。并学习如何使用单引号、双引号和反引号，以及它们的替代模式。我们将讨论 here 文档，以及如何在 CGI 脚本中使用它们。此外，读者还将学到如何借助警示信息和诊断信息对脚本中的错误进行排查。

练习 3 了解 Perl 的语法

1. 在命令行提示符下，编写打印如下内容的 Perl 语句。
   Hello world!!
   Welcome to Perl programming.
2. 执行另外一个 Perl 命令，打印 datebook 文件的内容。（读者可在华章网站（www.hzbook.com）上找到该文件。）
3. 执行 Perl 命令，显示当前使用的 Perl 版本和修订信息。
4. 将示例 3.1 示例程序复制到文本编辑器中，保存并检查它的语法是否正确，然后执行该程序。

第 4 章　获得打印句柄

4.1　文件句柄

在通常情况下，每当程序开始执行时，父进程（通常就是 shell 程序）便会打开三个预先定义的流，分别叫做 stdin、stdout 和 stderr。在默认情况下，这三个流都连接在终端屏幕上。

stdin 流是输入的来源，即终端键盘；stdout 是输出目的地，即屏幕；而 stderr 则是打印程序错误信息的地方，一般也是终端屏幕。

Perl 会从 shell 继承上述 stdin、stdout 和 stderr 流。Perl 并不直接访问这些流，而是把它们命名为文件句柄。Perl 只能通过这些文件句柄来访问上述流。其中，stdin 的文件句柄是 STDIN、stdout 的文件句柄是 STDOUT; 而 stderr 的文件句柄则是 STDERR。后面将详细介绍如何创建自己的文件句柄。读者现在只需使用上述预先定义好的句柄即可。

在默认情况下，print 和 printf 函数都会把输出发送到 STDOUT 文件句柄中。

4.2　字（Word）

在向 STDOUT 输出一系列字符内容时，读者最好先能理解 Perl 是如何处理这些字的。在 Perl 中，任何未加引号的字都必须以字母或数字开头，并由字母、数字或下划线组成。Perl 是区分字母大小写的。如果没有在字两边加上引号，就有可能与其他表示文件句柄的词、标记或其他保留字发生冲突。如果某个字在 Perl 中没有特殊含义的话，就应当把它放在单引号中。

4.3　print 函数

print 函数负责将字符串或由逗号隔开的字列表打印到 Perl 的 STDOUT 文件句柄中。如果调用成功，print 函数就返回 1，否则返回 0。

字符串常量 \n 可以出现在字符串的末尾，表示换行; 亦可嵌入到字符串中间位置，以便割裂该字符串。与 shell 一样，为了解释反斜杠，Perl 也要求将 \n 这样的转义序列置入到双引号中去。

示例 4.1

```
(The Script)
```

```
print "Hello", "world", "\n";
print "Hello world\n";

(Output)
Helloworld
Hello world
```

解释：

1. 每个传送给 print 函数的字符串都位于双引号之间，并以逗号分隔。如果需要打印空白字符，则必须把这些空白字符也放到引号中。为了表达换行符，还应当把 \n 转义序列也放入双引号之间。
2. 将双引号之间的整个字符串打印到标准输出。

示例 4.2

```
(The Script)
print Hello, world, "\n";

(Output)
No comma allowed after filehandle at ./perl.st line 1
```

解释：

如果没有给字符串加上引号，则必须指定 STDOUT 文件句柄。否则 Perl 便会把它碰到的第一个参数作为文件句柄的名字（譬如本例中的 Hello 字符串将被当作文件句柄）。在文件句柄后面不应跟随逗号；逗号只能用于待输出的各字符串之间。

示例 4.3

```
(The Script)
print STDOUT Hello, world, "\n";

(Output)
Helloworld
```

解释：

如果没有给字符串加上引号，则必须指定 STDOUT 文件句柄。如需解释 \n，则应当将它置于双引号之间。最好不要以这种方式使用不带引号的字符串。不在引号内的词又称为裸词(bareword)。

注：在 STDOUT 后面不应有逗号。

4.3.1 引号

不论用户使用 Perl 的什么功能，都可能碰到引号，尤其是在打印字符串的时候。一般而言，字符串都由两个配对的双引号或单引号包围起来。当字符串两头是单引号时，串内所有字符都被当作文本进行处理；而当两头是双引号时，尽管大部分字符都被当作纯文本，也会有少量字符是作为变量替换（variable substitiution）和特殊字符序列（special escape sequence）来处理的。本章将讨论特殊字符序列；而有关变量的内容请读者阅读第 5 章“变量”。

在 Perl 中，某些字符具有特殊含义，譬如美元符号（$）和 @ 符号。如果需要把这些符号当作纯文本来处理的话，就必须在它前面加上反斜杠（\），或者在它两边加上单引号（''）。其中，反斜杠只能处理单个字符，而不是整个字符串。

示例 4.4

```
(The Script)
1  $name="Ellie";
2  print "Hello, $name.\n";# $name and \n evaluated
3  print 'Hello, $name.\n';# String is literal; newline not
                           # interpreted

4  print "I don't care!\n";# \n is interpreted in double quotes
5  print 'I don\'t care!', "\n";# Backslash protects single quote
                                # in string "don\'t"
(Output)
2  Hello, Ellie.
3,4  Hello, $name.\nI don't care!
5 I don't care!
```

用户碰见最频繁的错误莫过于和引号相关的错误了。这里将介绍一些最常见的因为引号不匹配或者裸字情况而出现的出错信息。

读者不妨可以把引号想象成是 Perl 字符串的外套。如果脱了这层外套，就有可能收到下列与“BareWord”相关的信息：

Bareword“there”not allowed while“strict subs”in use at try.pl line3. Execution of program.pl aborted due to compilation errors.

读者也可把引号想象成是一对夫妻。一个双引号的配偶必然是另一个与之配对的双引号; 而一个单引号也只能和与之相配的单引号成为夫妻。如果没有对引号进行匹配的话，就好像半边天不见了，或者说“runaway”了。那剩下的另一半该何去何从呢？这时用户就会收到如下错误消息：

（Might be a runaway multi-line " " string starting on line 3）

触犯引号规则的情况

示例 4.5

```
(The Script)
    #!/usr/bin/perl
    # Program to illustrate printing literals
1   print "Hello, "I can't go there";  # Unmatched quotes
2   print "Good-bye";

(Output)
Bareword found where operator expected at qtest.plx line 2, near
""Hello, "I"
        (Missing operator before I?)
Bareword found where operator expected at qtest.plx line 3, near
"print "Good"
  (Might be a runaway multi-line "" string starting on line 2)
        (Do you need to predeclare print?)
String found where operator expected at qtest.plx line 3, at end of line
        (Missing semicolon on previous line?)
syntax error at qtest.plx line 2, near ""Hello, "I can't "
Can't find string terminator '"' anywhere before EOF at qtest.plx line 3
```

解释：

1. 字符串“Hello 的开头有一个双引号，但其尾部却没有与之配对的双引号。这种格式会造成麻烦的问题。Perl 会假定 I 前面的那个双引号是与上面这个引号配对的。结果造成后面另一个字符串“I can’t go there”又成了光杆字符串。而这一行末尾的双引号则会与下一行开头的双引号配对。总之一切都乱套了。
2. Perl 还会认为“Good_bye”也是光杆字符串，因为无法为它找到可供匹配的引号。第 1 行“there”末尾的双引号已经把本行“Good_bye”开头的引号配对过去了，从而造成“Good_bye”成了光杆字符串。

4.3.2　实量（常量）

当把某个实量值（literal value ⊖）赋予某个变量或在屏幕上打印它时，该实量可以表示某个数字。这些数字可以是十进制、八进制或者十六进制，也可以是用浮点模式或科学计数法表示的浮点数。

位于双引号之间的字符串中也可以含有实量（literal），譬如 \n 表示换行符，\t 表示制表符，\e 则表示取消（escape）。字符串实量是以一个反斜杠开头的位于字母表中的字符。它们可用于表达十进制、八进制、十六进制或控制符。

Perl 还提供了用于表示当前脚本名称、当前脚本行号和当前脚本逻辑末尾位置的实量。

鉴于读者可能会在 print 或 printf 函数中用到这些实量，不妨让我们在这里看看它们的庐山真面目。（若需获得更多有关常量定义的信息，请参阅附录 A 中的“常量”一栏。）

表 4-1　数字实量

示例	描述
12345	整数
0b1101	二进制数
0x456fff	十六进制数
0777	八进制数
23.45	浮点数
.234E-2	科学计数法

数字实量。数字实量能够以十进制、八进制或十六进制形式表示某个正整数或负数（详见表 4-1）。此外还能以浮点形式或科学计数法形式表达浮点数。其中，八进制整数必需以 0（零）开头，而十六进制数则必需以 0x（零和 x）开头。以科学计数法表达的数字必需在末尾带有一个 E，并在其后加上一个正数或负数以表示其指数情况。

字符串实量。和 shell 中的字符串相似，Perl 字符串也是通过单引号或双引号隔开的。在字符串中存在着所谓字符串实量（string literal），又称为逸出字符序列（escape sequence）。它必须出现在双引号之间，常用于解释反斜杠字符。

表 4-2　字符串实量

逸出序列	描述信息（ASCII 名称）
\t	制表符
\n	换行符
\r	回车符
\f	表格缩进

⊖ 实量又称为常量（Constant），但 Perl 方面的专家都爱用“实量（Literal）”这个词。本书为了向他们看齐，也将使用“实量”这个词。

（续）

逸出序列	描述信息（ASCII 名称）
\b	退格符
\a	警报 / 闹钟
\e	取消（escape）
\033	八进制字符
\xff	十六进制字符
\c[	控制符
\l	从下一字符开始切换为小写字母
\u	从下一字符开始切换为大写字母
\L	从下一字符开始到 \E 结束，切换为小写字母
\U	从下一字符开始到 \E 结束，切换为大写字母
\Q	在所有字符前追加反斜杠，直到碰到 \E
\E	\L 或 \U 的相应结束符
\\	反斜杠

示例 4.6

```
print "This string contains \t\ttwo tabs and a newline.\n" # Double quotes
(Output)
This string containstabs and a newline.

print 'This string contains\t\ttwo tabs and a newline.\n; #Single quotes
(Output)
This string contains\t\ttwo tabs and a newline.\n
```

特殊实量。Perl 提供了两个特殊实量 _LINE_ 和 _FILE_。它们往往用作分隔符，不论在单引号还是双引号中都不会解释。它们分别代表当前脚本的行数和名称。Perl 中的特殊实量等价于 C 语言中预定义的特殊宏（Macro）。

在脚本中，特殊实量 _END_ 负责表示文件的逻辑末尾位置。任何位于 _END_ 后面的内容都将被忽略，就好像它们成了注释一样。在 UNIX 中，表示文件末尾的控制序列是 <Ctrl>-d（\004），而在 MS-DOS 中则是 <Ctrl>-z（\032）；二者都等效于 _END_。

DATA 特殊实量则负责表示一个文件句柄，允许用户处理来自脚本的文本数据，而非外来数据。

示例 4.7

```
print "The script is called", _ _FILE_ _, "and we are on line number ",
_ _LINE_ _,"\n";
(Output)
The script is called ./testing.plx and we are on line number 2
```

注意：在特殊实量两边必须各提供两个下划线。（详见表 4-3）

表 4-3　特殊实量

实　量	描　述
_ _LINE_ _	表示当前行号
_ _FILE_ _	表示当前文件名
_ _END_ _	表示脚本的逻辑末尾位置；其后的内容都将被忽略
_ _DATA_ _	表示特殊文件句柄
_ _PACKAGE_ _	表示当前包；默认的包是 main

4.3.3　打印实量

在前面我们介绍了实量的表现形式，本节则探讨如何在 print 函数中使用这些实量（常量）。

打印数字实量

示例 4.8

```
(The Script)
    #!/usr/bin/perl
    # Program to illustrate printing literals
1   print "The price is $100.\n";
2   print "The price is \$100.\n";
3   print "The price is \$",100, ".\n";
4   print "The binary number is converted to: ",0b10001,".\n";
5   print "The octal number is converted to: ",0777,".\n";
6   print "The hexadecimal number is converted to: ",0xAbcF,".\n";
7   print "The unformatted number is ", 14.56, ".\n";
8   $now = localtime(); # A Perl function
9   $name = "Ellie"; # A string is assigned to a Perl variable
10  print "Today is $now, $name.";
11  print 'Today is $now, $name.';

(Output)
1   The price is.
2   The price is $100.
3   The price is $100.
4   The binary number is converted to: 17.
5   The octal number is converted to: 511.
6   The hexadecimal number is converted to: 43983.
7   The unformatted number is 14.56.
10  Today is Sat Mar 24 15:46:08 2007, Ellie.
11  Today is $now, $name.
```

解释：

1. 字符串 The price is $500 位于双引号中。其中，$ 是 Perl 规定的特殊字符，用于表示标量（参阅第 4 章“变量”），而不是表示货币。因此，由于程序中并没有定义 $100 变量，所以上述语句不会打印任何内容。由于单引号具有屏蔽所有字符解释的作用，因此若把 $100 置入单引号之间，便可避免它被当作变量处理。除此之外，在 $ 前面加上反斜杠也能起到相同的效果。不过，如果使用的是单引号的话，\n 也将成为普通的字符串，而不是表示换行。
2. 本行用反斜杠引用美元符号 $，故而会把它作为实量来处理。

3. 如果需要作为数字而非字符串进行处理的话，数字 100 就必须以单字（word）格式出现。这里即便没有跟随变量名，也必须对美元符号进行转义处理。若要解释为特殊字符，则必须将 \n 置于双引号之间。
4. 这是一个二进制数，因为它是以 0b（0 和 b）开头的。本行将打印其十进制值。
5. 这是一个八进制数，因为它是以 0（零）开头的。本行将打印其十进制值。
6. 这是一个十六进制数，因为它是以 0x（0 和 x）开头的。本行将打印其十进制值。
7. 这是一个数字，其打印形式和实际含义相同，都是 14.56。print 函数不会改变输出内容的格式。
8. Perl 拥有大量函数。读者已经学会了 print 函数。而 localTime() 则是 Perl 提供的另外一个函数。（函数名后面的括号是可有可无的。）该函数能够返回当前日期和时刻。这里把时间结果值赋给了一个名叫 $now 的 Perl 变量。在下一章中，读者将学习如何使用变量。
9. 本行把 $name 赋值为字符串“Ellie”。
10. 当字符串位于双引号之间时，print 函数会输出 $now 和 $name 这两个变量的值。
11. 当字符串位于单引号之间时，print 函数会把他们当作纯字符串予以输出。

打印字符串实量

示例 4.9

```
(The Script)
    #!/usr/bin/perl
1   print "***\tIn double quotes\t***\n";    # Backslash interpretation
2   print '%%%\t\tIn single quotes\t\t%%%\n'; # All characters are
                                              # printed as literals
3   print "\n";

(Output)
1   ***      In double  quotes         ***
2   %%%\t\tIn single quotes\t\t%%%\n
3
```

解释：

1. 如果字符串位于双引号之间，则执行反斜杠解释。其中，\t 是一个字符串实量，负责产生制表符；\n 则表示换行符。
2. 如果字符串位于单引号之间，则不解释特殊字符 \t 和 \n。将他们按原样打印。
3. 若要解释换行符 \n，则必需将其置于双引号之间。“\n”字符串将能实现换行。

示例 4.10

```
(The Script)
    #!/usr/bin/perl
1   print "\a\t\tThe \Unumber\E \LIS\E ",0777,".\n";

(Output)
1   (BEEP)          The NUMBER is 511.
```

解释：

\a 能够产生警报或蜂鸣音。后面紧跟两个 \t 制表符。\U 表示要以大写形式打印其后的字符串，直到碰到 \E 或行末为止。因此，这里的 number 将以大写形式打印出来，直到碰到 \E。而 \E 之前的字符串 IS 则会变为小写形式。本行接着打印八进制数字 0777 的值，最后打印句点和换行符。

打印特殊实量

示例 4.11

```
(The Script)
    #!/usr/bin/perl
    # Program, named literals.perl, written to test special literals
1   print "We are on line number ", _ _LINE_ _, ".\n";
2   print "The name of this file is ",_ _FILE_ _,".\n";
3   _ _END_ _
    And this stuff is just a bunch of chitter-chatter that is to be
    ignored by Perl.
    The _ _END_ _ literal is like Ctrl-d or \004.[a]

(Output)
1   We are on line number 3.
2   The name of this file is literals.perl.
```

解释：

1. 如需解释特殊实量 _LINE_，则应将其置于引号之间。该特殊实量负责保存 Perl 脚本的当前行号。
2. 当前脚本名是 literal.Perl。特殊实量 _FILE_ 能保存当前 Perl 脚本名。
3. 特殊实量 _END_ 表示脚本的逻辑终止位置，告诉 Perl 忽略其后出现的一切字符。

示例 4.12

```
(The Script)
    #!/usr/bin/perl
    # Program, named literals.perl2,
    # written to test special literal _ _DATA_ _
1   print <DATA>;
2   _ _DATA_ _
    This line will be printed.
    And so will this one.

(Output)
This line will be printed.
And so will this one.
```

解释：

1. print 函数将显示特殊实量 _DATA_ 下的所有文本内容。由于特殊实量 _DATA_ 是位于尖括号中的，因此 Perl 将把它作为文件句柄打开，print 函数将显示 <DATA> 读取的行内容。
2. 这是 <DATA> 文件句柄使用的数据（读者可以用 _END_ 代替 _DATA_，也能得到相同的结果）。

4.3.4 warning 编译指示符和 -w 开关

-w 开关能在用户使用潜在保留字或者其他可能导致程序出错的特性时发出警告。Larry Wall 在 Perl 5 的 man 主页中说："每当碰到无法解释的问题时，不妨试着打开 -w 开关！即便没有，最好也打开它。"

用户可使用 -w 作为 Perl 的命令行选项，譬如：

```
perl -w <scriptname>
```

或者在 Perl 脚本中的 shbang 行中指定，如：

```
#!/usr/bin/perl  -w
```

编译指示符（pragma）是一种特殊的 Perl 模块，负责告诉编译器如何编译语句块。用户可利用这种模块控制程序的行为。从 Perl 5.6.0 起，在标准 Perl 库中出现了 warning.pm 模块，其功能类似于 -w 开关。后者是一个编译指示符，用于控制警告类型。

用户亦可在程序的 #! 行下（若没有 #! 行则在脚本开头）添加下行内容：

```
use warnings;
```

这样就启用了所有可能的警告信息。若要关闭警告信息，只需在脚本中添加下列内容：

```
no warnings;
```

这样便可关闭脚本中所有可能的警告信息。

示例 4.13

```
    (The Script)
        #!/usr/bin/perl
        # Scriptname: warnme
        print STDOUT Ellie, what\'s up?;

(Output) (At the Command Line)
$ perl -w warnme
  Unquoted string "what" may clash with future reserved word at warnme line 3.
  Backslash found where operator expected at warnme line 3, near "what\"
  Syntax error at warnme line 3, near "what\"
  Can't find string terminator "'" anywhere before EOF at warnme line 3.
```

解释：

-w 开关（详见附录 A）负责打印那些标识符意义不明情况的警告信息，譬如只使用了一次的变量，不正确的字符串和数字转换等。由于字符串 Ellie 没有加引号，因此 Perl 会错误地认为它是保留字或未定义的文件句柄。本例中其他的错误信息则是由字符串中不匹配的引号所引发的。

示例 4.14

```
    (The Script)
        #!/usr/bin/perl
        # Scriptname: warnme
        use warnings;
        print STDOUT Ellie, what\'s up?;

(Output)
Unquoted string "what" may clash with future reserved word at warnme line 3.
Backslash found where operator expected at warnme line 3, near "what\"
Syntax error at warnme line 3, near "what\"
Can't find string terminator "'" anywhere before EOF at warnme line 3.
```

解释：

Perl 5.6 和其后出现的版本都使用 warning 编辑指示符代替了 -w 开关。其中，use 函数负责使用那些位于 Perl 库中的模块。warning 编译指示符则负责发送有关标识符二义性的警告信息，由于字符串 Ellie 没有加引号，因此 Perl 会错误地认为它是保留字或未定义的文件句柄。由于该程序没有闭合引号以结束字符，因此编译器会发出警告。

4.3.5 diagnostics 编译指示符

该编译指示符除了能够显示警告信息外，还能提供有关当前错误的更详细解释。与 warning 编译指示符类似，它只能影响脚本的编译阶段。但和前者不同的是，它将假定读者是不懂得编程的新手，并提供尽可能详细的出错信息。

示例 4.15

```
(The Script)
use diagnostics;
print "Hello there';  # Unmatched quote
print "We are on line number ", _ _LINE_ _,"\n";

(The output)
Bareword found where operator expected at test.plx line 3, near "$now
= "Ellie"
  (Might be a runaway multi-line "" string starting on line 2) (#1)
    (S syntax) The Perl lexer knows whether to expect a term or an
operator.
    If it sees what it knows to be a term when it was expecting to see
    an operator, it gives you this warning.  Usually it indicates that
    an operator or delimiter was omitted, such as a semicolon.

         (Missing operator before Ellie?)
String found where operator expected at test.plx line 3, at end of
line  (#1)
         (Missing semicolon on previous line?)

syntax error at test.plx line 3, near "$now = "Ellie"
Can't find string terminator '"' anywhere before EOF at test.plx line
3 (#2)
    (F) Probably means you had a syntax error.  Common reasons
include:

        A keyword is misspelled.
        A semicolon is missing.
        A comma is missing.
        An opening or closing parenthesis is missing.
print "hello there';
print "We are on line number ", _ _LINE_ _,"\n";
```

解释：

Perl 5.6 及其后续版本使用 diagnostics 编译指示符代替了 -w 开关和 warning 编译指示符。这个特殊的 Perl 模块将发送有关脚本出错详情的消息。由于这个脚本中的字符串 Hello 不含匹配的引号，因此 diagnostics 编译指示符会列出造成问题的所有潜在因素。编译器将要求该字符串以另一个匹配的引号结尾。

4.3.6 strict 编译指示符

本节将讨论的是 strict 编译指示符。当使用该指示符时，一旦程序违反了相关约束条件，就会编译不通过。如上面例子所示，如果出现了裸字符[⊖]（即未加引号的字符），则 strict 编译指示符就不能通过，从而造成程序退出。用户可通过各种参数控制 strict 编译指示符的内容（完整的列表内

⊖ 为字符加上引号就好比为它穿上衣服。如果去掉引号，字符就好像裸露了一样。

容请参阅附录 A)。

示例 4.16

```
(The Script)
    #!/usr/bin/perl
    # Program: stricts.test
    # Script to demonstrate the strict pragma
    use strict "subs";
    $name = Ellie;              # Unquoted word Ellie
    print "Hi $name.\n";

(Output)
$ stricts.test
  Bareword "Ellie" not allowed while "strict subs" in use at
./stricts.test line 5.
  Execution of stricts.test aborted due to compilation errors.
```

解释:

通过 use 函数便可使用 Perl 标准库中的模块。当 strict 编译指示符以 subs 作为参数时，便会在内部编译时捕获程序中任何位置的裸字。一旦发现有裸字，就立刻退出程序并报错。

4.4 printf 函数

printf 函数负责将格式化的字符串输出到选定的文件句柄。其默认文件句柄是 STDOUT。它和在 C、awk 中使用的 printf 函数功能相同。如果 printf 函数执行成功，其返回值是 1，否则为 0。

printf 函数后面带有一个加有引号的负责格式规范的控制字符串。其后则是一系列由逗号隔开的参数，这些参数都必须是简单表达式。其格式说明符以 % 开头。而对于每个以 % 开头的格式说明符，必须有一个参数与它对应（详见表 4-4 和表 4-5）。

用户也可选择在字符串和表达式两边加上小括号。

示例 4.17

```
printf("The name is %s and the number is %d\n","John",50);
```

解释:

1. 待打印的字符串位于双引号之间。其第一个格式说明符是 %s，对应于参数 John，直接定位在第一个逗号的右侧。跟在百分号 % 后面的 s 也叫转换字符(conversion character)。s 说明该处将进行字符串转换。在本例中，John 将在打印输出时替换其中的 %s。
2. %d 格式则说明十进制（整数）值 50 将在字符串中打印出来。

表 4-4 格式化转换符

转换符	定 义
%b	无符号二进制整数
%c	字符
%d、i	十进制整数
%e	科学计数法浮点数
%E	使用大写字母 E 的科学计数法浮点数
%f、%F	浮点数
%g	使用 e 或 f 转换符的浮点数，取其最小宽度
%G	使用 e 或 f 转换符的浮点数，取其最大宽度
%id、%D	长整型十进制数
%lu、%U	无符号长整型十进制数
%lo、%O	长整型八进制数
%p	指针（十六进制数）
%s	字符串
%u	无符号的十进制数
%x	十六进制数
%X	使用大写字母 X 的十六进制数
%lx	长整型十六进制数
%%	打印百分号实量

标记修饰符位于 % 之后，负责进一步定义输出的格式。例如，%-20s 表示输出长度为 20 个字符的左对齐的字符串。

表 4-5 标记修饰符

转换符	定 义
%-	左对齐修饰符
%#	如果是八进制数，则在显示时带上前导 0；如果是十六进制数，则在显示时带有前缀 0x
%+	对于使用 d、e、f 和 g 的转换符，显示其整数部分，并显示正负号＋、－
%0	把显示内容中的空白部分以 0 补足
%number	最大字段宽度。譬如，若 number 为 6（如 %6d），说明最大字段宽度是 6
%.number	指定浮点数的精度。例如，%.2f 表示小数点后两位 ；%8.2 表示最大字段宽度为 8，并精确到小数点后两位

在打印参数时，字段（field）负责为打印结果提供位置，其宽度是包含在字段中的字符数。字

段的宽度由百分号 % 与代表字段最大宽度的数字来决定，其后紧跟着相应转换字符。譬如，%20s 表示字段宽度为 20 个字符的右对齐的字符串；%-25s 表示字段宽度为 25 个字符的左对齐的字符串；%10.2f 则表示字段宽度为 10 个字符的浮点数（其中小数点算一个字符），并精确到小数点后 2 位。如果参数宽度超过了最大字段宽度的话，printf 也不会对数字进行截取，只不过其输出格式可能变得较为难看。譬如，如果需要截去小数点右边的数字的话，printf 将对该数字进行四舍五入。如格式转换符为 %.2f，相应参数为 56.55555，则会打印出 56.6。

示例 4.18

```
(The Script)
    #!/usr/bin/perl
1   printf "Hello to you and yours %s!\n","Sam McGoo!";
2   printf("%-15s%-20s\n", "Jack", "Sprat");
3   printf "The number in decimal is %d\n", 45;
4   printf "The formatted number is |%10d|\n", 100;
5   printf "The number printed with leading zeros is |%010d|\n", 5;

6   printf "Left-justified the number is |%-10d|\n", 100;
7   printf "The number in octal is %o\n",15;
8   printf "The number in hexadecimal is %x\n", 15;
9   printf "The formatted floating point number is |%8.2f|\n",
       14.3456;
10  printf "The floating point number is |%8f|\n", 15;
11  printf "The character is %c\n", 65;

(Output)
1   Hello to you and yours Sam McGoo!
2   Jack                   Sprat
3   The number in decimal is 45
4   The formatted number is |       100|
5   The number printed with leading zeros is |0000000005|.
6   Left-justified the number is |100       |
7   The number in octal is 17
8   The number in hexadecimal is f
9   The formatted floating point number is |    14.35|
10  The floating point number is |15.000000|
11  The character is A
```

解释：

1. 位于引号之间的字符串含有 %s 格式转换符。在打印时用字符串 Sam Mcgoo 替换 %s。
2. 字符串 Jack 的字段宽度为 15 个字符，并且是左对齐的。字符串 Sprat 的字段宽度为 20 个字符，也是左对齐的。其中的括号是可选的。
3. 数字 45 将以十进制整数格式打印。
4. 数字 100 的字段宽度是 10，并且是右对齐的。
5. 数字 5 的字段宽度是 10，它是右对齐的，其前缀是先导零（0）而非空格。如果在表示字段宽度的数字前放置修饰符 0 的话，就表示在打印时必须用先导零补全字段前部的空白部分。
6. 数字 100 的字段宽度为 10，左对齐。
7. 字段 15 以八进制整数形式打印。
8. 数字 15 以十六进制整数形式打印。
9. 数字 14.3456 的字段宽度为 8 个字符，包括一个小数点字符。其可选部分表示精确到两位小数并进行四舍五入。

10. 数字 15 的字段宽度是 8 个字符，右对齐，默认精确到小数点后 6 位。
11. 数字 65 将转换为 ASCII 字符 A 并打印出来。

4.4.1 sprintf 函数

sprintf 函数与 printf 函数基本类似，所不同的是前者允许给变量赋予格式化字符串。sprintf 函数和 printf 函数使用相同的转换符表（详见表 4-4 和表 4-5）。有关变量的内容将在第 5 章“变量”中予以详细介绍。

示例 4.19

```
(The Script)
1   $string = sprintf("The name is: %10s\nThe number is: %8.2f\n",
                      "Ellie", 33);
2   print "$string";

(Output)
2   The name is:       Ellie
    The number is:     33.00
```

解释：

1. sprintf 函数遵循和 printf 相同的字符、字符串和数字转换规则。其真正的区别仅限于 sprintf 允许把格式化后的输出内容保存到变量中。在本例中，格式化后的输出内容将存储于标量型变量 $string 里。字符串中插入的 \n 会导致换行的发生。有关标量的详细内容将在第 5 章“名字里的乾坤”予以详细介绍。本行的小括号是可选的。
2. 打印出来的变量值是 sprintf 函数产生的格式化输出。

4.4.2 无引号打印：here 文档

Perl 的 here 文档（here document）特性直接来自于 UNIX shell 中的 here 文档。它允许引用一整块文档内容，这些内容必须位于名叫“用户自定义终止符（user-defined terminator）”的字段之间。在第一个终止符到最后一个终止符之间出现的文本就相当于加上了引号，换而言之就是为“从 here 到 here”的本文加上引号。here 文档是一种面向行的使用格式，其格式是：首先提供一个起始的终止符和分号，然后提供 << 运算符。<< 运算符后面可能没有空格，除非其终止符本身也需要被引用。如果终止符的两边没有加上单引号或双引号的话，则执行变量扩展。如果终止符两边有单引号的话，则不执行变量扩展。本文的每一行将插入到第一个和最后一个终止符之间。最终终止符必须自己独占一行，并且周围不能出现空白字符。

与 shell 不同的是，Perl 在 here 文档中不能执行命令替换（即备份引用，backquotes）操作。此外，当终止符出现在备份引用内容中时，Perl 也能执行 here 文档中的命令。

在 CGI 脚本里 here 文档常用于提供大块的 HTML 标签内容。

示例 4.20

```
(The Script)
1   $price=1000;    # A variable is assigned a value.
2   print <<EOF;
3   The consumer commented, "As I look over my budget, I'd say
```

```
4   the price of $price is right. I'll give you \$500 to start."\n
5   EOF

6   print <<'FINIS';
    The consumer commented, "As I look over my budget, I'd say
7   the price of $price is too much.\n I'll settle for $500."
8   FINIS

9   print << x 4;
    Here's to a new day.
    Cheers!
10
    print "\nLet's execute some commands.\n";
    # If terminator is in backquotes, will execute OS commands
11  print <<'END';
    echo Today is
    date
    END

(Output)
3   The consumer commented, "As I look over my budget, I'd say
    the price of 1000 is right. I'll give you $500 to start."
6   The consumer commented, "As I look over my budget, I'd say
    the price of $price is too much. \n I'll settle for $500."
9   Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
11  Let's execute some commands.
    Today is
    Fri Oct 27 12:48:36 PDT 2007
```

解释：

1. 标量型变量 $price 的值是 1000。
2. 开始 here 文档操作。其中 EOF 表示文件终止符。如果它出现在双引号中，Perl 便会将它作为文本块进行处理。如果在终止符前面存在任何空白字符，则应当把终止符放在双引号之间，如：“EOF”。
3. 引用 here 文档主体中的所有文本块，其效果就像所有的文本内容都位于双引号之间一样。
4. 当出现在双引号之间时，美元符号 $ 就拥有了特殊含义。由于 here 文档中出现的文本效果与出现在双引号中的一样，因此其变量在此也具有特殊含义。$ 符号表示正在使用标量型变量。这会引起变量值的解析操作。如果美元符号前面带有反斜杠 \ 的话，Perl 则把它当作实量进行处理。如果在这里使用特殊的反斜杠序列如 \n 的话，Perl 也将一样解析它们。
5. here 文档的结尾由匹配的终止符标记，这里是 EOF。
6. 通过把终止符 FINIS 置于单引号之间，随后出现在程序文本中的特殊字符（如美元符号 $ 或反斜杠序列）的特殊含义都将关闭。
7. 效果等同于位于单引号之间的文本。
8. 闭合（closing）终止符，标志着 here 文档的结束。

9. 值 x4 表明 here 文档中的文本将打印 4 次。运算符 x 又称作是重复运算符。在文本块的末尾必须出现一个空行，以便使 here 文档顺利结束。
10. 在这个 here 文档的末尾要求提供一个空行。
11. 终止符出现在备份引用（backup reference）中。Shell 能够执行位于 END 和 END 之间的命令。本例中包含了 UNIX 命令。如果使用其他操作系统如 Windows 或 Mac OS 的话，其命令必须与所在操作系统兼容。

here 文档与 CGI。下面的程序又称作是 CGI（公共网关接口）程序，它是由 Web 服务器而非 shell 执行的一种简单 Perl 程序。它和一般的 Perl 程序相同，主要区别在于如下两点：

1. 它有一行名叫 MIME（即 Context-type:text/html）的内容，负责描述发送回浏览器的内容类型。
2. 文档中提供了嵌入 HTML 标签的文本，以便浏览器用不同的颜色、字体和类型来呈现文本的语言构成。很多 CGI 程序员都使用 here 文档，以免对程序的每一行都进行 print。

CGI 程序存储于特殊的 cgi-bin 目录下，用户一般可在服务器的根目录下面找到它。有关 CGI 的更多知识请参阅第 16 章“CGI 和 Perl”。

若要执行如下脚本，请启动 Web 浏览器，并在地址框中输入：http://servername/cgi-bin/scriptname⊖。详见图 4-1。

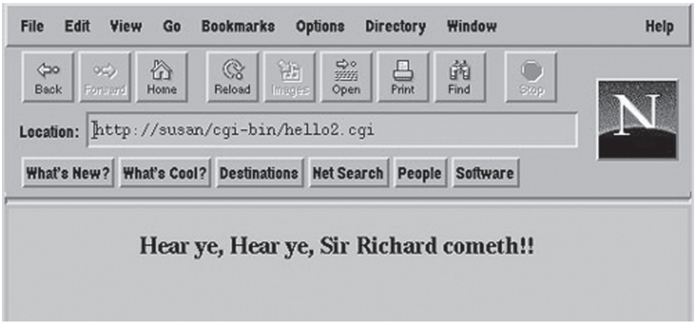


图 4-1　示例 4.21 相应的 Web 浏览器视图

示例 4.21

```
    #!/bin/perl
    # The HTML tags are embedded in the here document to avoid using
    # multiple print statements
1   print <<EOF;      #here document in a CGI script
2   Content-type: text/html
3
4   <HTML><HEAD><TITLE>Town Crier</TITLE></HEAD>
    <H1><CENTER>Hear ye, hear ye, Sir Richard cometh!!</CENTER></H1>
    </HTML>
5   EOF
```

解释：

1. here 文档从本行开始，其终止符是 EOF。Print 函数将接收位于 EOF 和 EOF 之间的所有内容。
2. 这一行告诉浏览器将要发送的内容类型是混合有 HTML 标签的文本内容。本行的末尾必须提供一个空行。
3. 文档体内容，其中含有文本内容和 HTML 标签。
4. 字符 EOF 负责标识文档的结束位置。

⊖ 用户必须在此提供正确的服务器名和正确的脚本名。请注意，某些 CGI 文件必须提供 .cgi 或 .pl 扩展名。

4.5 读者应当学到的知识

1. 如何定义 stdin、stdout 以及 stderr？
2.“文件句柄”一词的含义是什么？
3. 如何以八进制格式表示一个数字？十进制呢？十六进制呢？
4. print 和 printf 函数的主要区别是什么？
5. 在处理字符串时，单引号和双引号有什么区别？
6. 什么是“实量（Literal）”？
7. _END_ 的用途是什么？
8. 什么是反斜杠序列？
9. sprintf 函数的功能是什么？
10. 什么是编译指示符（progma）？
11. 如何检查代码以确保语法的正确性？
12. 什么是 here 文档？如何在 CGI 程序中使用它？

4.6 下章简介

在下一章内容中，读者将学到有关变量和“特殊标志（funny symbol）”的知识。读者将会创建和访问标量、数组、散列上下文以及命名空间。读者还能从用户处获取输入，并了解为何需要对其进行“截取（chomp）”。此外，下一章还会介绍几种不同的数组和散列函数。

练习 4 Perl 字符串练习

1. 请使用 print 函数输出如下字符串：

"Ouch," cried Mrs. O'Neil, "You musn't do that Mr. O'Neil!"

2. 请使用 printf 函数将 $34.6666666 输出成 $34.67。
3. 请编写一段名为 literal.plx 的 Perl 脚本，打印如下内容：

```
$ perl literals
Today is Mon Mar 12 12:58:04 PDT 2007 (Use localtime())
The name of this PERL SCRIPT is literals.
Hello. The number we will examine is 125.5.
The NUMBER in decimal is 125.
The following number is taking up 20 spaces and is right justified.
|                        125|
                The number in hex is 7d
                The number in octal is 175
The number in scientific notation is 1.255000e+02
The unformatted number is 125.500000
The formatted number is 125.50
My boss just said, "Can't you loan me $12.50 for my lunch?"
I flatly said, "No way!"
```

```
Good-bye (Makes a beep sound)
```

4. 向上述文件脚本中添加一个 here 文档，用于打印：

   Life is good with Perl.

   I have just completed my second exercise!

5. 读者应当如何在脚本中打开警告功能？又该如何开启诊断功能呢？

第 5 章　变　　量

5.1　Perl 变量简介

在开始阅读本章内容之前，请读者留意，本章提供的每段代码都是经过编号的。代码后面提供的说明则依据代码编号一一给出。这样做的意图是让读者充分理解每个程序中的重要代码行。在把这些代码复制到自己的文本编辑器时，请读者不要复制那些编号，否则代码会执行出错！

5.1.1　类型

在任何一种编程语言中，变量都是最为基础的概念。变量是一种数据项，其值可随着程序的运行而发生变化。与之相反，直接量和常量的值则是固定不变的。变量可以位于程序的任何位置，并且不需要像在高级语言中那样声明它为特定的类型。用户可以把字符串、数字或它们的组合赋值给 Perl 变量。例如可以在某个变量中存储一个数字，然后再改变为存储一个字符串。Perl 都来者不拒。

Perl 变量主要有三种类型：标量型、数组型以及关联数组型（常称为散列 [Hash]）。其中，标量型变量含有单个值；数组型变量含有一列有序的值，并可通过正整数形式的下标予以索引；而散列型变量则含有无序的键 / 值对（Key Value Pair）集合，由字符串（即它的键）作为索引，关联到相应的值上。（详见下文的“变量、数组和散列”一节内容）。

5.1.2　作用域和包

变量的作用域决定了它在程序中的哪些位置是可见的。在 Perl 脚本中，变量是对整个程序可见的（即其作用域是全局的），并可在程序中的任何位置予以改变。

在 Perl 内部，前面章节所述的 Perl 程序范例都会编译到所谓的包（Package）中，以便为变量提供命名空间。几乎所有的 Perl 变量在包中都是全局的。全局变量是在整个包范围内都可见的，如果在包中某个位置修改了该全局变量，则这个更改操作将永久地影响该变量。默认的包名字叫做 main，类似于 C 语言中的 main 函数。这种变量对应于 C 程序中的静态变量。在 Perl 中，用户无需关心 main 包的命名问题，以及它在编译期间的处理方式。这里之所以提到包的概念，是为了让读者了解变量的作用域其实是在 main 包内的，也就是全局的。后面在讨论 main 包中的 our、local 和

my 函数时，读者会发现其实也可以改变变量的作用域和命名空间。



图 5-1 main 包里标量、列表和散列的命名空间

5.1.3 命名规范

与 C 和 Java 不同的是，Perl 变量无需声明便可使用。Perl 中的变量拥有自己的命名空间。它们是由作为前缀的特殊字符（funny characters）进行标识的。其中，标量型变量的前缀是 $，数组型变量的前缀是 @，而散列型变量的前缀则是 %。由于特殊字符表明了该变量的类型，因此用户可以对标量、数组和散列型变量分别赋予相同的变量名而不必担心名字冲突。例如，$name、@name 和 %name 属于不同类型的变量，第一个是标量型，第二个是数组型，第三个是散列型[⊖]。

由于保留字和文件句柄前面没有相应的特殊字符前缀，因此变量名也不会和保留字或文件句柄冲突。变量名是大小写敏感的。因此，$Num、$num 和 $NUM 表示的是不同的变量。

如果一个变量是以字母开头的，则它可以由任意数目的字母（下划线也算作字母）或数字构成。如果变量名不以字母开头的话，它就只能由一个字符构成。Perl 提供了一组特殊的变量（如 $_、$^、$.、$1、$2 等），它们就属于上述第二种类型（详见附录 A 中的“特殊变量”一节）。在特殊情况下，变量的前缀还可以是单引号，但这种情况只有在使用包时才可能出现。

尚未初始化的变量值是 0 或者 null，具体初始值视其上下文是数字还是字符串而决定。

5.1.4 赋值语句

赋值运算符也就是等于号（＝），常常用于将其右侧的值赋予左侧的变量。任何可以“赋”的值都代表了一个已经命名的存储空间，称为左值（lvalue[⊜]）。如果位于赋值运算符左侧的内容不是左值的话，Perl 便会报错。

在给变量赋值时，如果等于号左侧的变量是标量型的话，Perl 会在标量上下文中计算等于号右侧的表达式。如果等于号左边的变量是一个数组，Perl 则会在数组上下文中计算右边的表达式（详见“标量、数组和散列”一节）。

简单语句是指以分号结尾的表达式。

⊖ Perl 允许使用相同的变量名，但不推荐这么做，因为这样可能会造成代码阅读时的困惑。

⊜ 等于号左边的值称为左值（lvalue），等于号右侧的值称为右值（rvalue）。

格式

```
variable = expression ;
```

示例 5.1

```
(The Script)
     # Scalar, array, and hash assignment
1    $salary=50000;                     # Scalar assignment
2    @months=('Mar', 'Apr', 'May');     # Array assignment
3    %states= (                         # Hash assignment
        'CA' =>  'California',
        'ME'  => 'Maine',
        'MT'  => 'Montana',
        'NM'  => 'New Mexico',
             );
4    print "$salary\n";
5    print "@months\n";
6    print "$months[0], $months[1], $months[2]\n";
7    print "$states{'CA'}, $states{'NM'}\n";
8    print $x + 3, "\n";               # $x just came to life!
9    print "***$name***\n";            # $name is born!
(Output)
4    50000
5    Mar Apr May
6    Mar, Apr, May
7    California, New Mexico
8    3
9    ******
```

解释：

1. 为标量型变量 $salary 赋值 50000（详见“标量型变量”一节）。
2. 为数组 @months 赋予以逗号隔开的列表：Mar，Apr，May。该列表包含在括号中，并且表中每一项都加上了引号（详见“数组”一节）。
3. 散列型变量 %states 赋值为由连字符（=>）或逗号隔开的一组字符串组成的列表[a]。左侧的字符串称为键[b]（Key），右侧的字符串称为值（Value）。键及其相应值之间存在相互关联的关系（详见“散列”一节）。
4. 打印标量 $salary 的值，然后换行。
5. 打印数组 @months 的内容。借助双引号预留各个元素之间的空格位置。
6. 数组 @months 中的单个元素属于标量，因此其前缀应当是美元符号（$）。数组是从零开始索引的。
7. 散列变量 %states 的键元素位于花括号（{}）中。本行将打印其相关联的值。每个散列值都是独立的标量值，以美元符号（$）作前缀。
8. 这里首次引用了标量型变量 $x。由于上下文是数字型的，因此本行代码将把 $x 加上 3。其中，$x 的原始值是 0。
9. 这里首次引用了标量型变量 $name。其上下文是字符串型的，因此其初始值是 null。

a. 在 Perl 4 和 Perl 5 中都可以使用逗号，而 => 连字符则是由 Perl 5 引进的。

b. 与逗号不同，=> 连字符用于给键（key）加引号。如果一个键由多个字符组成的话，则无需为其加引号。

5.1.5　引号规范

由于引号影响着解释变量的方式，因此这里有必要回顾一下 Perl 的引号规则。Perl 的引号规则

与 shell 相似。对于 shell 程序员而言，这可不是什么好消息，因为他们对于使用引号往往感觉很灰心，经常难以决定使用哪一种引号，在何处使用才好，而且在碰到引号相关错误时也难以找到问题所在。换句话说，引号是真正的调试噩梦⊖。对于这类人，Perl 也提供了另一种加引号的方法⊜。

Perl 拥有三种类型的引号，均提供不同的功能。它们分别是单引号、双引号和反引号（backquote）。

反斜杠的行为类似于一组单引号，但它只能用于引用单个字符。

单引号和双引号可用于界定字符串。根据所用引号类型的不同，引号可以解释特殊字符，也可用于保护字符免于解释。

单引号是一种“民主”的引号。它对位于其中的所有字符均一视同仁。换而言之，单引号不支持特殊字符，而双引号则会区别对待一般字符和特殊字符。后者会将字符串中的部分字符当作特殊字符对待。其特殊字符包括美元符号 $、@ 符号以及转义序列，如 \t 和 \n 等。

当命令位于反引号中时，该命令将由 shell 执行。这种特性又称为命令置换。该命令的输出内容可作为 print 函数的输出内容，也可赋给某个特定变量。如果使用的操作系统是 Windows、Linux 或 UNIX 的话，位于反引号中的命令必须是具体操作系统支持的命令。

无论用户使用哪一种引号，都必须成对匹配使用。由于引号负责标识字符串的起始或结束，因此如果忘记了其中某个引号的话，Perl 会报“Might be a multilane runaway string”或者“Execution of quotes aborted”或者“Can’t find terminator anywhere before EOF...”错误，并停止编译。

双引号。除非出现在单引号中间，否则双引号必须成对匹配出现。

当字符串位于双引号之间时，Perl 将允许解释标量型变量（以 $ 开头）和数组型变量（以 @ 开头），即字符串中的变量名将由相应变量值替换。不过双引号之间的字符串不能出现散列变量（以 % 开头）。

如果字符串中含有字符串型实量（如 \t、\n），则必须带有反斜杠，并位于双引号之间。

在双引号之间也可出现单引号，譬如“I don’t care!”。

示例 5.2

```
(The Script)
    # Double quotes
1   $num=5;
2   print "The number is $num.\n";
3   print "I need \$5.00.\n";
4   print "\t\tI can't help you.\n";

(Output)
2   The number is 5.
3   I need $5.00.
4           I can't help you.
```

解释：

1. 为标量型变量 $num 赋值为 5。
2. 字符串位于双引号之间。打印标量型变量的值，并解释字符串实量 \n。
3. 当以反斜杠 \ 开头时，美元符号会作为实量打印出来，即忽略变量置换。
4. 当出现在双引号之间时，解释特殊符号 \t 和 \n。

单引号。如果字符串出现在单引号之间，则直接打印其实量内容即可（所见即所得）。

如果在字符串中需要单引号，则可以将其嵌入双引号中，或者放在反斜杠后面。如果需要把双

⊖ 在 Barrg Rosenberg 的 KornShell Programming Tutorial 中专门有一章的名字就叫“The Quotes From Hell”。

⊜ Larry Wall 是 Perl 的创始人之一，他称这种替换途径为“语法宝贝”。

引号当作实量对待，也可将其嵌入单引号中。

示例 5.3

```
(The Script)
    # Single quotes
1   print 'I need $100.00.', "\n";
2   print 'The string literal, \t, is used to represent a tab.', "\n";
3   print 'She cried, "Help me!"', "\n";

(Output)
1   I need $100.00.
2   The string literal, \t, is used to represent a tab.
3   She cried, "Help me!"
```

解释：

1. 美元符号 $ 将解释为直接量。在双引号中，美元符号也可解释为标量型变量的起始字符。为了解释反斜杠，则需要把 \n 置于双引号之间。
2. 这里不会把字符串型直接量 \t 解释为制表符，而是以实量形式直接输出。
3. 出现在单引号之间的这个双引号将被如实输出（即以实量形式打印）。

反引号。位于反引号之间的 UNIX/Windows [1]命令将由 shell 负责执行，并将其输出内容返回给 Perl 程序。这些输出内容一般会赋给某个变量，或者直接作为 print 函数的输出部分。当把命令输出内容赋予变量时，其上下文语境必须是标量（即赋予单个值）[2]。为了支持命令置换，用户不能把反引号置于单引号或者双引号之间（UNIX shell 程序员请注意，与 shell 程序一样，不能把反引号放在双引号之间）。

示例 5.4

```
(The Script for UNIX/Linux)
    # Backquotes and command substitution
1   print "The date is ", 'date';       # Windows users: 'date /T'
2   print "The date is 'date'", ".\n"; # Backquotes treated literally
3   $directory='pwd';                   # Windows users: 'cd'
4   print "\nThe current directory is $directory.";

(Output)
1   The date is Mon Jun 25 17:27:49 PDT 2007.
2   The date is 'date'.
4   The current directory is /home/jody/ellie/perl.
```

解释：

1. 本行的 date 命令由 UNIX shell 执行，其输出内容将返回给 Perl 的 print 函数。date 命令的输出内容中含有换行符。
2. 当反引号位于单引号或者双引号之间时，就不会发生命令置换。
3. 把 UNIX pwd 命令的输出内容（如当前工作目录）赋予标量型变量 $dir。对于 Windows 用户而言，相同功能的命令则是“cd”。
4. 将标量 $dir 的值输出到屏幕上。

---

[1] 如果使用其他的操作系统如 DOS 或 Mac OS 9.1 的话，可供使用的 OS 命令也相应不同。

[2] 如果将命令的输出内容赋予数组的话，输出的第一行将成为数组的第一个元素，第二行会成为数组的第二个元素，依此类推。

Perl 的替换符号。 Perl 提供了另一种表示引号的形式——即 q、qq、qx 和 qw 结构。

- q 代表一个单引号。
- qq 代表一个双引号。
- qx 代表一个反引号。
- qw 代表引用的字列表。（详见“数组”一节）。

表 5-1　引号替换结构

引用结构	含义
q/Hello/	‘Hello’
qq/Hello/	“Hello”
qx/date/	‘date’
@list=qw/red yellow blue/;	@list={ ‘red’ , ‘yellow’ , ‘blue’ }

其中，被引用的字符串位于正斜杠（/）之间，其定界符可适用于所有 4 种 q 结构。定界符可以是单个字符，也可是成对字符。

```
>q/Hello/
q#Hello#
q{Hello}
q[Hello]
q(Hello)
```

示例 5.5

```
(The Script)
    # Using alternative quotes
1   print 'She cried, "I can\'t help you!"',"\n";  # Clumsy
2   print qq/She cried, "I can't help you!"\n/;    # qq for double
                                                   # quotes
3   print qq(I need $5.00\n);  # Really need single quotes
                               # for a literal dollar sign to print
4   print q/I need $5.00\n/;   # What about backslash interpretation?
    print qq(I need \$5.00\n); # Can escape the dollar sign
5   print qq/\n/, q/I need $5.00/,"\n";
6   print q!I need $5.00!,"\n";
7   print "The present working directory is ", 'pwd';
8   print qq/Today is /, qx/date/;
9   print "The hour is ", qx{date +%H};

(Output)
1   She cried, "I can't help you!"
2   She cried, "I can't help you!"
3   I need .00
4   I need $5.00\nI need $5.00

5   I need $5.00
6   I need $5.00
7   The present working directory is /home/jody/ellie/perl
8   Today is Mon Jun 25 17:29:34 PDT 2007
9   The hour is 17
```

解释：

1. 这里的字符串位于单引号之间。这样就允许以实量的形式输出其中的普通引号。位于 can\’t 中的单引号由反斜杠引用，因此它也将作为实量进行输出。如果没有反斜杠的话，它就必须与第一个单引号进行匹配。如果最后一个单引号没有响应的匹配项，程序就会告诉用户出现了失控的引号，或者查找其匹配项直到文件末尾。
2. qq 用于代表双引号。正斜杠则负责界定字符串的起始范围。
3. 由于使用了 qq，$5.00 中的美元符号（$）将解释为含有空值的标量型变量。因此本行将输出 .00（这可不是对待钱的好办法）。
4. 单个 q 代表单引号。$5 将作为实量处理。不过 \n 也会被当作实量。在单引号中间，是不解释反斜杠的。
5. \n 位于带有 qq 结构的双引号之间，而字符串 I need $5.00 则是使用 q 结构的单引号引用。其中第二个 \n 使用老式的双引号形式。
6. 惊叹号（！）是一个替换了的界定符，它和 q 结构一起使用，负责界定字符串的起止位置。
7. 字符串 The present working directory is 位于双引号中；UNIX 命令 pwd 则位于反引号之间，用于命令置换。
8. qq 结构引用了 Today is 字符串；qx 结构则用于替换反引号，用于进行命令置换。
9. 花括号是一个替换的界定符，它（而不是斜杠）与 qx 结构一起使用。本行将打印 UNIX date 命令的输出内容。

5.2　标量、数组和散列

前文介绍了 Perl 变量的基本概念（包括其类型、可见性、特殊字符等），下文将更详细地对它们进行介绍。在学习了本节内容后，读者将对变量的引用机制有更深入的了解，并了解到引号是如何影响 Perl 程序解释过程的。

5.2.1　标量型变量

标量是以 $ 开头的单个数字或者字符串。在引用标量或给标量赋值时，必须为其加上 $ 前缀。如果读者已经熟悉了 shell 编程的话，一定会对赋值时也使用美元符号感到不习惯。

赋值。在赋值时，Perl 会把等号右边的值作为单值进行运算（即其上下文语境是标量）。即便其中含有很多字符，系统还是会把加了引号的字符串当作标量来处理。

示例 5.6

```
$number = 150;
$name = "Jody Savage";
$today = localtime();
```

解释：

1. 为标量型变量 $number 赋予数字值 150。
2. 字符串 Jody Savage 将作为单值字符串赋值给标量 $name。
3. Perl 函数 localTime 的输出内容将以字符串形式赋值给 $today。

花括号。如果一个标量位于花括号（{ }）之间的话，其后面出现的字符串将不会追加到标量后面。

示例 5.7

```
(The Script)
1   $var="net";
2   print "${var}work\n";

(Output)
3   network
```

解释：

1. 为标量 $var 赋值为 net。
2. 变量位于花括号中，以表达它与后面的 work 字符串的区别。如果不使用花括号的话，本行将不会输出任何内容，因为程序中没有定义名为 $varwork 的变量。

示例 5.8

```
(The Script)
    # Initializing scalars and printing their values
1   $num = 5;
2   $friend = "John Smith";
3   $money = 125.75;
4   $now = localtime();         # localtime() is a Perl function
5   $month="Jan";
6   print "$num\n";
7   print "$friend\n";
8   print "I need \$$money.\n";     # Protecting our money
9   print qq/$friend gave me \$$money.\n/;
10  print qq/The time is $now\n/;
11  print "The month is ${month}uary.\n";     # Curly braces shield
                                              # the variable
12  print "The month is $month" . "uary.\n"; # Concatenate
(Output)
6   5
7   John Smith
8   I need $125.75.
9   John Smith gave me $125.75.
10  The time is Sat Jan 24 16:12:49 2007.
11  The month is January.
12  The month is January.
```

解释：

1. 给标量 $num 赋值为 5。
2. 给标量 $friend 赋予字符串值 John Smith。
3. 给标量 $money 赋予浮点数字值 125.75。
4. 将 Perl 函数 localtime() 的输出内容赋予标量 $now。
5. 将标量 $month 赋予字符串值 Jan。
6. 打印标量 $sum 的值。
7. 打印标量 $friend 的值。
8. 打印位于引号之间的字符串。其中的反斜杠使得第一个美元符号（$）以实量形式打印；然后，将 $money 的值插入到双引号中，并打印它的值。

9. 本行以 Perl 的 qq 结构代替了双引号。其中被引用的字符串位于正斜杠之间。这里还插入了 $friend 的值，并在 $money 的值之前加上了 $ 前缀。
10. 打印位于引号之间的字符串，其中插入了 $now 变量值。
11. 和在 UNIX shell 中一样，花括号会让变量字符免受后面跟随的其他字符的干扰。本行将输出 January。
12. 读者还可借助点号（.）操作符把两个不同的字符串链接到一起（详见第 6 章内容），这项功能又称作串联（concatenation）。

defined 函数。如果一个标量既不是有效的字符串，又不是有效的数字值的话，就称它是未定义的。defined 函数可以检查变量值的有效性。如果变量中含有值，该函数返回值为 1，否则就返回 null。该函数还可用于检查数组、子例程和 null 字符串的有效性。

示例 5.9

```
$name="Tommy";
print "OK \n" if defined $name;
```

undef 函数。该函数用于为已经定义的变量解除其定义。它能够释放那些分配给变量的内存空间。该函数将返回未定义的值，并能释放与数组或子例程相关联的内存空间。

示例 5.10

```
undef $name;
```

$_ 标量型变量。$_ 是随处可见的一个字符。尽管它在 Perl 中也很有用，但用户却很少能看到它。在执行搜索时，它可以用作默认的模式空格符，并负责保有当前行号。一旦为 $_ 赋予了某个数值，则诸如 chomp、split 和 print 之类的函数就会把新的 $_ 值当作输入参数。读者将在后面学到更多有函数及其参数的知识。不过在这里，读者可以先看看如下这段代码：

示例 5.11

```
1   $_ = "Donald Duck";
2   print;    # The value of $_ is printed
    Donald Duck
```

解释：

1. 本行向 $_ 标量型变量中赋予字符串值“Donald Duck”、
2. print 函数没有收到任何待打印的内容，因此它会输出 $_ 的值，后者是没有其他字符串参数情况下的默认输出。

5.2.2　数组

如果用户需要保存一组相似的数据元素的话，那么使用数组要比为每个元素单独创建变量来得更方便。用户可以借助数组名把单独的变量名与数据元素列表关联起来，然后通过数组名与下标引用列表中的每一个元素。

与 C 语言不同，Perl 并不要求数组元素具有相同的数据类型。数组元素可以是数字和字符串的混合体。在 Perl 中，数组是一个有名字的列表，含有有序的标量型变量集合。数组名必须以 @ 开

头。数组名的后面是位于方括号（[ ]）之间的下标，它们都是非负整数，从 0 开始计数。

赋值。在初始化数组时，用户可以把所有元素放在括号中，并以逗号分隔它们。这里要为列表加上括号，是因为逗号运算符的优先级低于赋值运算符。数组中的所有元素必须是标量型变量。

Perl 5 引入了 qw 结构，它也能用于创建列表（类似于 qq、q 和 qx）。列表中各项内容将作为带有单引号的字符串来处理。

```
$pal = "John"; # Scalar holds one value
@pals = ( "John", "Sam", "Nicky", "Jake" ); # Array holds a list of values
```

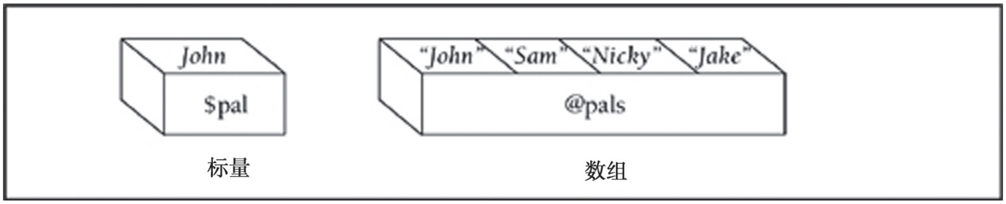


图 5-2　标量和数组

示例 5.12

```
1   @name=("Guy", "Tom", "Dan",  "Roy");
2   @list=(2..10);
3   @grades=(100, 90, 65, 96, 40, 75);
4   @items=($a, $b, $c);
5   @empty=();
6   $size=@items;
7   @mammals = qw/dogs cats cows/;
8   @fruit = qw(apples pears peaches);
```

解释：

1. 使用一个含有 4 个字符串实量的列表对数组 @name 进行初始化。
2. 为数组 @list 赋值，其内容是从 2 到 10 的所有整数（参见示例 5.14）。
3. 使用一个含有 6 个数字实量的列表对数组 @grades 进行初始化。
4. 使用一个含有 3 个标量型变量的列表对数组 @items 进行初始化。
5. 使用一个空列表对 @empty 数组进行初始化。
6. 将数组 @items 赋值给一个标量型变量 $size。该标量的值将是数组中的元素个数（在本例中，其值等于 3）。
7. qw（引用字）结构后面跟随选定的定界符。列表中每个字都作为带有单引号的字来对待。末尾的定界符说明列表到此结束。本例亦可写作：
   `@mammals = { ' cats ' , ' dogs ' , ' cows ' }`
8. qw 结构支持任何成对的字符如 ()、{}、<> 和 [] 作为定界符。

特殊标量和数组赋值。特殊标量 $#arrayname 能返回数组中最后一个元素的下标。由于数组下标是从 0 开始的，因此该值将比数组长度小 1。$#arrayname 还可用于缩短或者截取数组的长度。

$[变量表示当前数组的初始下标 0。用户可以改变这个变量，从而使得下标从 1 而不是 0 开始。但是，Larry Wall 并不鼓励这样做。

示例 5.13

```
1   @grades = (90,89,78,100,87);
    print "The original array is: @grades\n";
2   print "The number of the last index is $#grades\n";

3   $#grades=3;
    print "The array is truncated to 4 elements: @grades\n";

4   @grades=();
    print "The array is completely truncated: @grades\n";

(Output)
  The original array is: 90 89 78 100 87
  The number of the last index is 4
  The array is truncated to 4 elements: 90 89 78 100
  The array is completely truncated:
```

解释：

1. 把一个含有 5 个数字的列表赋予 @grades 数组。
2. $# 结构能够获得数组最后一个元素的下标（即索引值）。
3. 将数组下标缩减为 3。
4. 本行使用了一个空列表，使得该数组被完全清空为空列表。

范围运算符和数组赋值。　当用在数组上下文语境中时，运算符 .. 又称为范围运算符，负责从左到右逐个返回一组值。

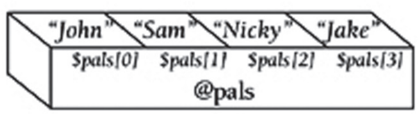
示例 5.14

```
1   @digits=(0 .. 10);
2   @letters=( 'A' .. 'Z' );
3   @alpha=( 'A' .. 'Z', 'a' .. 'z' );
4   @n=( -5 .. 20 );
```

解释：

1. 把从 0 到 20 的数字列表赋予数组 @digits。
2. 把从 A 到 Z 的大写字母列表赋予数组 @letter。
3. 把所有大写字母和小写字母的列表赋予数组 @alpha。
4. 把从 −5 到 20 的数字列表赋予数组 @n。

访问元素。如要引用数组中的某个元素，用户必须把每个元素都看成是标量（即以美元符号 $ 作为前缀），并且从 0 开始计算下标。数组下标都是整数。例如，在数组 @names 中，第一个元素是 @names[0]，下面一个是 @names[1]，依此类推。



数组元素

图 5-3　数组元素

示例 5.15

```
(The Script)
    # Populating an array and printing its values
1   @names=('John', 'Joe', 'Jake');     # @names=qw/John Joe Jake/;
2   print @names, "\n";  # prints without the separator
3   print "Hi $names[0], $names[1], and $names[2]!\n";
4   $number=@names;      # The scalar is assigned the number
                         # of elements in the array
5   print "There are $number elements in the \@names array.\n";
6   print "The last element of the array is $names[$number - 1].\n";
7   print "The last element of the array is $names[$#names].\n";
                         # Remember, the array index starts at zero!!
8   @fruit = qw(apples pears peaches plums);
9   print "The first element of the \@fruit array  is $fruit[0];
          the second element is $fruit[1].\n";
10  print "Starting at the end of the array; @fruit[-1, -3]\n";

(Output)
2   JohnJoeJake
3   Hi John, Joe, and Jake!
5   There are 3 elements in the @names array.
6   The last element of the array is Jake.
7   The last element of the array is Jake.
9   The first element of the @fruit array is apples; the second element is pears.
10  Starting at the end of the array: plums pears
```

解释：

1. 向数组 @names 赋予三个字符串：John、Joe 和 Jake。
2. 将整个数组打印到 STDOUT 中，忽略其中的空格。
3. 从下标 0 开始打印数组的每个元素。
4. 把数组 @names 的元素个数值赋予标量型变量 $number。
5. 打印数组 @names 中的元素个数。
6. 打印数组的最后一个元素。由于数组下标是从 0 开始的，因此只要把数组长度减去 1 就能得到数组中最后一个元素的下标值。
7. 打印数组的最后一个元素。$#names 表示数组最后一个元素的下标值。本行使用该变量直接检索数组 @names 中的最后一个元素。
8. qw 结构可用于创建字数组，同时无须将这些字包含在引号中或用逗号分隔。这里的定界符可以是任意匹配的字符对（详见“替换引号”一节）。
9. 打印 @fruit 数组的前两个元素。
10. 当使用负数作为下标时，将从数组末端选择数组元素。其中，最后元素 ($fruit[−1]) 是 plums，而倒数第三个元素 ($fruit[−3]) 则是 pears。请注意，当索引值位于同一组括号中时，譬如 ($fruit[−1,−3])，则表示引用列表而非标量。这也是为什么要在数组名前面带有 @ 而非 $ 符号。

数组分片（Array Slices）。当把某个数组的值赋予到另一个数组中时，得到的数组称之为数组分片。

如果位于赋值运算符右侧的数组长度大于左边的数组，Perl 会把多余的那部分元素丢弃；否则，则将剩余的元素赋值为未定义值。如下面的示例所示，位于数组片中的索引不必是连续编号；只需为每个元素指定赋值运算符右侧的相应数组元素即可。

示例 5.16

```
(The Script)
    # Array slices
1   @names=('Tom', 'Dick', 'Harry', 'Pete' );
2   @pal=@names[1,2,3];  # slice -- @names[1..3] also O.K.
3   print "@pal\n\n";

4   ($friend[0], $friend[1], $friend[2])=@names;    # Array slice
5   print "@friend\n";

(Output)
3   Dick Harry Pete

5   Tom Dick Harry
```

解释：

1. 把元素值‘Tom’、‘Dick’、‘Harry’和‘Pete’赋予数组 @names。
2. 把数组 @names 元素 1、2 和 3 赋予数组 @pal。本行会把数组 @names 的元素分片存储到 @pal 数组中。
3. 通过抽取数组 @names 的分片元素 1、2、3，创建数组 @pal。

示例 5.17

```
(The Script)
    # Array slices
1   @colors=('red','green','yellow','orange');
2   ($c[0], $c[1],$c[3], $c[5])=@colors;  # The slice
3   print "**********\n";
4   print @colors,"\n";  # Prints entire array, but does
                         # not separate elements quoted
5   print "@colors,\n";  # Prints the entire array with
                         # elements separated
6   print "**********\n";
7   print $c[0],"\n";    # red
8   print $c[1],"\n";    # green
9   print $c[2],"\n";    # undefined
10  print $c[3],"\n";    # yellow
11  print $c[4],"\n";    # undefined
12  print $c[5],"\n";    # orange
13  print "**********\n" ;
14  print "The size of the \@c array is ", $#c + 1,".\n";

(Output)
3   **********
4   redgreenyelloworange
5   red  green yellow orange
6   **********
7   red
8   green
9
10  yellow
11
12  orange
13  **********
14  The size of the @c array is 6.
```

解释：

1. 将元素‘red’、‘green’、‘yellow’和‘orange’赋予数组 @colors。
2. 创建数组分片，它由四个标量型元素组成：$c[0]、$c[1]、$c[3] 和 $c[5]。请注意：这个数组分片中的下标编号不是连续的。
3. 为清楚起见，这里打印一行星号。
4. 在打印时不分开各个元素。
5. 当数组位于双引号之间时，会保留元素之间的空白字符。
6. 打印另一行星号。
7. 打印数组分片的第 1 个元素 red。
8. 打印数组分片的第 2 个元素 green。
9. 数组分片的第 3 个元素值是未定义的。由于没有给它赋值，所以它的内容是空值。
10. 打印数组分片的第 4 个元素 yellow。
11. 数组分片的第 5 个元素未定义。
12. 打印数组分片的第 6 个元素 orange。
13. 打印另一行星号。
14. 即便 @c 数组中的某些元素是未定义的，但通过该数组的容量，仍旧能说明这些未定义元素是存在的。

多维数组：列表的列表。多维数组有时也称作是表或者矩阵。它由行和列构成，并通过多重下标予以表示。在一个二维数组中，第一个下标代表行号，第二个下标则表示列号。

在 Perl 中，二位数组的每一行都包含在方括号之间。每一行都是一个无名列表（unnamed list），又称匿名数组（anonymous array），由其本身的元素组成。箭头运算符（arrow）又称为中缀（infix）运算符，用于获取数组中的单个元素。相邻的方括号之间等效于隐含了一个箭头运算符。（第 13 章将详细讨论匿名变量。）

示例 5.18

```
(The Script)
   # A two-dimensional array consisting of 4 rows and 3 columns
    @matrix=( [ 3 , 4, 10 ],     # Each row is an unnamed list
              [ 2,  7, 12 ],
              [ 0,  3,  4 ],
              [ 6,  5,  9 ],
            ) ;
    print "@matrix\n";
    print "Row 0, column 0 is $matrix[0][0].\n";
                 # can also be written - $matrix[0]->[0]
    print "Row 1, column  0 is $matrix[1][0].\n";
                 # can also be written - $matrix[1]->[0]
    for($i=0; $i < 4; $i++){
        for($x=0; $x < 3; $x++){
           print "$matrix[$i][$x]  ";
        }
        print "\n";
    }

(Output)
```

```
2    ARRAY(0xbf838) ARRAY(0xc7768) ARRAY(0xc77a4) ARRAY(0xc77e0)
3    Row 0, column 0 is 3.
4    Row 1, column 0 is 2.
7    3 4 10
     2 7 12
     0 3 4
     6 5 9
```

解释：

1. 把四个无名或匿名数组赋给数组 @matrix，其中每个匿名数组中含有 3 个元素值。
2. 打印 4 个匿名数组的地址。如要访问匿名数组中的单个元素，则必须使用其双重下标，或者使用箭头运算符。
3. 打印数组 @matrix 中第一个匿名数组的第一个元素值。-> 符号又称作箭头或者中缀运算符，用于分隔对数组或者散列的引用。@matrix[0][0] 或 $matrix[0]->[0] 表示第一行的第一个元素，其下标是从 0 开始的。
4. 打印 @matrix 第 2 行的第 1 个元素。其中，$matrix[1]->[0] 是 @matrix[1][0] 的另一种表达方式。
5. 外部 for 循环负责从第 0 行开始遍历每一行。在 for 循环体第一次执行完毕后，进入第二个 for 循环。
6. 内部 for 循环的速度要比外部循环快。它负责打印指定行的每一个元素值，然后将控制权返回给外部循环。
7. 打印矩阵中的每个元素。其中第一个下标代表行数，而第二个下标则代表列号。

5.2.3　散列

关联数组（associative array）又称作散列。它是由成对的标量（譬如字符串、数字或布尔量）所构成的。其中第一组标量必须与第二组标量相关联。在字符串对中，第一个字符串称为键（key），第二个字符串则称为值（value）。请留意，数组是有序的列表，其下标是从 0 开始顺序排列的；而散列表则是无序的，其字符串下标是随机分布的。（因此请不要指望在打印散列表时能按照当初输入的顺序得到输出。）

散列是一组无序的键 / 值对（key/value pair）。它类似于一张数据表，其中左侧是键，右侧则是与这些键相关联的值。散列的名字必须以 % 开头。

```
% pet = ( "Name"  => "Sneaky",
          "Type"  => "Cat",
          "Owner" => "Carol",
          "Color" => "yellow",
        );
```

键	值
“Name”	“Sneaky”
“Type”	“Cat”
“Owner”	“Carol”
“Color”	“yellow”

散列表

赋值。在引用元素前，必须首先定义散列。由于散列是由键 / 值对组成的，通过其第一个字符串予以索引，因此一旦键值对中的元素没有出现在数组定义里面，就会对键及其相应值造成影响。在为键和值赋值时，请确保该键和对应的值是关联的。在索引散列时，应当使用花括号，而不是方括号。

示例 5.19

```
1   %seasons=("Sp" => "Spring",
              "Su" => "Summer",
              "F"  => "Fall",
              "W"  => "Winter",
             );
2   %days=("Mon" => "Monday",
           "Tue" => "Tuesday",
           "Wed" => undef,
          );
3   $days{"Wed"}="Wednesday";
4   $days{5}="Friday";
```

解释：

1. 向散列 %seasons 赋予键和值。其中每个键与值都通过连字运算符 => 隔开。字符串 Sp 是一个键，其相应的值则是 Spring；字符串 Su 是另一个键，其相应值则是 Summer，等等。如果是单个字的话，用户无需给键加引号。
2. 向散列 %days 赋予键和值。其中向第三个键 Wed 赋值 undef。undef 函数等效于空值字符串。
3. 散列内的单个元素是标量型的。本行将向 Wed 键赋字符串值 Wednesday。其下标包含于花括号中。
4. 向键 5 赋字符串值 Friday。请注意，键是没有连续编号的，而键 / 值对则可通过序号和 / 或字符串构成。

访问元素。在访问散列中的值时，其下标由位于花括号中的键构成。Perl 提供了一组函数，能够列出散列的键、值以及每个元素（详见“数组函数”一节）。

由于使用了内部散列技术来存储键，因此在打印散列内容时 Perl 不保证其顺序性。

示例 5.20

```
(The Script)
    # Assigning keys and values to a hash
1   %department = (
                   "Eng" => "Engineering",
                   "M"   => "Math",
                   "S"   => "Science",
                   "CS"  => "Computer Science",
                   "Ed"  => "Education",
3                 );
4   $department = $department{'M'}; # Either single or double quotes
                                 # ok for the keys
5   $school = $department{'Ed'};
6   print "I work in the $department section\n" ;
7   print "Funds in the $school department are being cut.\n";
8   print qq/I'm currently enrolled in a $department{'CS'} course.\n/;
9   print qq/The department hash looks like this:\n/;
```

```
10  print %department, "\n";    # The printout is not in the expected
                                # order due to internal hashing

(Output)
6   I work in the Math section
7   Funds in the Education department are being cut.
8   I'm currently enrolled in a Computer Science course.
9   The department hash looks like this:
10  SScienceCSComputer ScienceEdEducationMMathEngEngineering
```

解释：

1. 散列的名字是 %department。赋予它键和值。
2. 第一个键是字符串 Eng，和它关联的值则是 Engineering。
3. 结尾的括号和分号负责结束赋值操作。
4. 把值 Math 赋予标量 $ department，并将它和键 M 相关联。
5. 把值 Education 赋予标量 $ school，并将它和键 Ed 相关联。
6. 打印位于引号内的字符串，并插入标量 $ departmen 的值。
7. 打印位于引号内的字符串，并插入标量 $ school 的值。
8. 打印位于引号内的字符串以及与键 CS 相关联的值。
9. 打印位于引号内的字符串。
10. 将键和值放在一起打印整个散列表，发现其打印顺序不同于事先的预期。

散列分片（hash slices）。 所谓散列分片，是一组散列键的列表，并且这些键的关联值是另一组键的列表。该列表由多个散列名组成，并以 @ 符号开头。散列键的列表则位于花括号中。

示例 5.21

```
(The Script)
    # Hash slices
1   %officer= ("NAME"=> "Tom Savage",
               "SSN" => "510-22-3456",
               "DOB" => "05/19/66"
              );

2   @info=qw(Marine Captain 50000);
3   @officer{'BRANCH', 'TITLE', 'SALARY'}=@info;
    # This is a hash slice
4   @sliceinfo=@officer{'NAME','BRANCH','TITLE'};
    # This is also a hash slice
5   print "The new values from the hash slice are: @sliceinfo\n\n";
    print "The hash now looks like this:\n";
6   foreach $key ('NAME', 'SSN', 'DOB', 'BRANCH', 'TITLE', 'SALARY'){
7      printf "Key: %-10sValue: %-15s\n", $key, $officer{$key};
    }

(Output)
5   The new values from the hash slice are: Tom Savage Marine Captain

    The hash now looks like this:
7   Key: NAME Value: Tom Savage
    Key: SSN Value: 510-22-3456
    Key: DOB Value: 05/19/66
```

```
    Key: BRANCH Value: Marine
    Key: TITLE Value: Captain
    Key: SALARY Value: 50000
```

解释：

1. 赋予散列 %officer 相应的键和值。
2. 把三个值赋予数组 @info。
3. 这是一个散列分片的实例。本行把散列 officer 对应的键 BRANCH、TITLE、SALARY 赋值为一个数组，该数组含有 Marine、Captain 和 50000 这三个元素。散列的名字必须以 @ 开头，因为它是一组键的列表，并能接受另一组值列表 @info 的相应值。
4. 在赋值时，本行通过散列分片创建了一个名叫 @sliceinfo 的数组。该数组由关联到键 BRANCH、TITLE 和 SALARY 的散列值组成。
5. 打印所创建的散列分片的值。
6. 借助 foreach 循环遍历键列表。
7. 打印键及其相应值的内容。分片在第三行将创建新的键 / 值对。

5.2.4 复杂数据结构

如果把数组和散列结合起来使用，用户便能得到一些更为复杂的数据结构，譬如散列数组、内嵌散列的散列表、含有数组的数组，等等。为了创建这些数据结构，读者必须首先了解 Perl 指针的工作原理。本节将通过几个示例介绍这方面的部分知识。有关如何创建复杂数据类型的详细内容将在第 13 章详细介绍。

散列的散列。散列中可以含有其他的散列。这就和在记录中嵌套记录的情况一样。被嵌套的散列没有名字，即为匿名散列，可通过箭头运算符予以引用。在有名字的散列中，每个键对应的值本身也是一个散列。匿名散列则由其本身的键 / 值对组成。

示例 5.22

```
(The Script)
    # Nested hashes
                                                values
                  keys             key      value     key      value
1   %students=( "Math"     => { "Joe"  => 100,    "Joan" => 95 },
                "Science" => { "Bill" => 85,     "Dan"  => 76 }
          );
2   print "On the math test Joan got ";
3   print qq/$students{Math}->{Joan}.\n/;
4   print "On the science test Bill got ";
5   print qq/$students{Science}->{Bill}.\n/;

(Output)
3   On the math test Joan got 95.
5   On the science test Bill got 85.
```

解释：

1. 散列 %student 由两个键组成：Math 和 Science。与这些键相关联的值则位于花括号中，并含有嵌套的键 / 值对。Math 键的相应值中含有两个嵌套的键：Joe 和 Joan，其值分别是 100 与 95。Science 键的相应值中则含有另外两个嵌套的键：Bill 和 Dan，其值分别是 85 和 76。

所有嵌套的键和值都位于一个匿名的散列表中。
2. 箭头运算符 ->（或中缀运算符）用于访问散列 %students 中匿名散列内的值。

示例 5.23

```
(The Script)
    # Anonymous arrays as keys in a hash
1   %grades=("Math"    => [ 90, 100, 94 ],
             "Science" => [ 77, 87, 86 ],
             "English" => [ 65, 76, 99, 100 ],
            );
2   print %grades, "\n";
3   print "The third math grade is: $grades{Math}->[2]\n";
4   print "All of the science grades are: @{$grades{Science}}\n";

(Output)
2   EnglishARRAY(0x8a65128)ScienceARRAY(0x8a650b0)MathARRAY(0x8a6f134)
3   The third math grade is: 94
4   All of the science grades are: 77 87 86
```

解释：

1. 赋予散列 %grades 相应的键和值。其中的值是匿名的数字列表。Perl 知道这是匿名列表，因为它们的值都位于方括号中。这些匿名数组存储于内存中，用户可通过箭头标记访问这些内存单元。
2. 本行将每个键的相应值打印为十六进制的地址，并以保存在该位置的类型 ARRAY 开头。
3. 通过把箭头运算符放在散列键之后，本行检索了第一个列表中的第 3 个元素。由于列表下标是从 0 开始，因此 $[Math]->[2] 指向数组的第 3 个元素。
4. 如需访问 Science 程序的整个列表，用户应当把键 / 值对置于花括号中，并以 @ 符号为前缀字符。

散列数组。数组可以含有嵌套的散列，正如在数组中嵌套记录一样。其中，数组的每个元素都是匿名的散列表，具有一组键以及相应的值。

示例 5.24

```
(The Script)
    # An array of hashes
1   @stores=( { "Boss" =>"Ari Goldberg",
                "Employees" => 24,
                "Registers" => 10,
                "Sales" => 15000.00,
              },
2             { "Boss" =>"Ben Chien",
                "Employees" => 12,
                "Registers" => 5,
                "Sales" => 3500.00,
              },
    );
3   print "The number of elements in the array: ",
4       $#stores + 1, "\n";   # The number of the last subscript + 1

5   for($i=0; $i< $#stores + 1; $i++){
6      print $stores[$i]->{"Boss"},"\n";    # Access an array element
       print $stores[$i]->{"Employees"},"\n";
```

```
        print $stores[$i]->{"Registers"},"\n";
        print $stores[$i]->{"Sales"},"\n";
        print "-" x 20 ,"\n";
    }

(Output)
3   The number of elements in the array: 2
6   Ari Goldberg
    24
    10
    15000
    --------------------
    Ben Chien
    12
    5
    3500
    --------------------
```

解释：

1. 数组 @stores 含有两个散列，一个是 Ari Goldberg 的商店，另一个则是 Ben Chien 的商店。该数组的每个元素值都是散列。
2. 这是 @stores 数组的第二个元素，它是一个散列。
3. $#stores 能够获得 @stores 数组中最后一个下标的数值。由于下标是从 0 开始的，因此把它加上 1 即可得到数组中的元素总数，这里是 2。
4. 若要访问数组中某个元素的值，首先需要在散列中规定下标数字，然后规定键。这里不要求必须使用箭头运算符，但是使用的话能让程序可读性更好。

5.3 从 STDIN 读取输入

请读者回忆一下，文件句柄 STDIN、STDOUT 和 STDERR 分别是三个预定义流 stdin、stdout 和 stderr 的名称。在默认情况下，这三个文件句柄都与终端屏幕相连。在打印输出到屏幕时，可使用 STDOUT；打印出错信息时，使用 STDERR；而在将用户输入内容赋予变量时，则可使用 STDIN。

如果使用 Perl 的输入运算符 <> 包围 STDIN 文件句柄，则程序会从终端键盘上读取标准输入，并保存到某个变量里。与 shell 和 C 读取输入的操作不同，Perl 在从标准输入读取各行内容时会保留行尾的换行符。如果程序不需要换行，则必须显式地删除或截除（chomp）掉换行符（详见“chop 与 chomp 函数”一节）。

5.3.1 把输入内容赋值给标量型变量

当从文件句柄 STDIN 读取输入内容时，倘若上下文语境是标量型，程序将每次读取输入的一行（包括换行符），并将其内容作为单个字符串赋予标量型变量。

示例 5.25

```
(The Script)
    # Getting a line of input from the keyboard.
1   print "What is your name?  ";
```

```
2   $name = <STDIN>;
3   print "What is your father's name? ";
4   $paname=<>;
5   print "Hello respected one, $paname";

(Output)
1   What is your name? Isabel
3   What is your father's name? Nick
5   Hello respected one, Nick
```

解释：

1. 把字符串 What is your name? 发送给 STDOUT 文件句柄，该句柄默认指向终端屏幕。
2. 位于输入运算符 <>（又称方块运算符）中的 STDIN 读取一行输入，并把这一行内容及其末尾的换行符赋值给变量 $name。在把输入内容赋予标量型变量时，程序会一直读取输入字符直到用户按下 Enter 键。
3. 把字符串打印到 STDOUT 文件句柄。
4. 如果输入运算符是空的，则从 STDIN 读取下一行输入内容，其行为与上述第 2 步相同，只不过这次是将输入值赋予 $paname。

5.3.2　chop 与 chomp 函数

chop 函数负责删除标量型变量的最后一个字符或数组中每个元素的最后一个字符，并返回修改后的值。chop 一般用于删除程序接收到的输入行末尾的换行符，这些输入行可以来自 STDIN、文件或者命令置换结果。对于 Perl 初学者来讲，位于行末的换行符确实很让人头疼。

Perl 5 中引入了 chomp 函数，负责删除标量型变量中的最后一个字符，或者数组中每个字的最后一个字符，并保证只有当该行末字符是换行符（或者更精确地说，是代表输入行分隔符的字符，该字符初始值是换行符，存储于 $/ 变量中）时才进行删除操作。它会返回删除后的字符数目。使用 chomp 函数来代替 chop，就能有效避免删除换行符之外的其他有用字符。

示例 5.26

```
(The Script)
    # Getting rid of the trailing newline. Use chomp instead of chop.
1   print "Hello there, and what is your name? ";
2   $name = <STDIN>;
3   print "$name is a very high class name.\n";
4   chop($name);    # Removes the last character no matter what it is.
5   print "$name is a very high class name.\n\n";
6   chop($name);
7   print "$name has been chopped a little too much.\n";
8   print "What is your age?  ";
9   chomp($age=<STDIN>); # Removes the last character if
                         # it is the newline.
10  chomp($age);         # The last character is not removed
                         # unless a newline.
11  print "For $age, you look so young!\n";

(Output)
1   Hello there, and what is your name? Joe Smith
3   Joe Smith
```

```
    is a very high class name.
5   Joe Smith is a very high class name.

7   Joe Smit has been chopped a little too much.

8   What is your age? 25
11  For 25, you look so young!
```

解释：

1. 把引用字符串打印到屏幕，默认就是 STDOUT 句柄。
2. 把用户输入的单行文本内容赋予标量型变量。<> 运算符用于读取输入。在这里，它是从 STDIN 文件句柄（即终端键盘）中读取的。在赋予 $name 的文本内容末尾存在一个换行符。
3. 打印 $name 的值。请注意，换行符断开了用户输入中 Joe Smith 之后的内容。
4. chop 函数负责删除赋予 $name 的字符串的最后一个字符，并返回删去的字符。
5. 在执行 chop 函数后再次打印字符串内容。其最后一个字符（在这里即为换行符）已经删除。
6. 本次 chop 操作将删除 Joe Smith 的最后一个字符，即 Smith 中的 h。
7. 将引用的字符串打印到 STDOUT 文件句柄中，可见已经删除了最后一个字符。
8. 本行代码首先把用户输入赋值给变量 $age。然后删除末尾的换行符，也就是保存在特殊变量 $/ 中的字符，其默认值就是换行符。最后返回删去的字符数。由于赋值运算符（=）的优先级较低，所以本行通过括号保证在 chomp 函数删除之前予以赋值。
9. 第二个 chomp 函数将不起作用。本行代码将仅仅删除末尾的换行符。这样做要比使用 chop 更加安全。
10. 打印 chomp 后的变量字符串。

5.3.3 read 函数

read 函数⊖可以从指定的文件句柄中将指定数目的字节读入到变量里。如果是从标准输入中读取的话，相应的文件句柄就是 STDIN。read 函数将返回读到的字节数目。

格式：

```
number_of_bytes = read(FILEHANDLE,buffer,how_many_bytes);
```

示例 5.27

```
(The Script)
    # Reading input in a requested number of bytes
1   print "Describe your favorite food in 10 bytes or less.\n";
    print "If you type less than 10 characters, press Ctrl-d on a line
           by itself.\n";
2   $number=read(STDIN, $favorite, 10);
3   print "You just typed: $favorite\n";
4   print "The number of bytes read was $number.\n";

(Output)
1   Describe your favorite food in 10 bytes or less.
    If you type less than 10 characters, press Ctrl-d on a line by
```

⊖ read 函数类似于 C 语言里的 fread 函数。

```
        itself.
        apple pie and ice cream          <-user input
3       You just typed: apple pie
4       The number of bytes read was 10.
```

解释：

1. 本行首先要求用户提供输入。如果输入的字符数小于 10 个，则应按下 <Ctrl>-d 退出。
2. read 函数带有多个参数：第一个参数是 STDIN，表示输入源；第二个参数是标量型变量 $favorite，用于保存输入内容；第三个参数则是要读取的字符数（按字节算）。
3. 打印读入的前 10 个字符，并舍弃其余的字符。
4. 实际读取的字符数（按字节算）将存储在 $number 中，并打印出来。

5.3.4　getc 函数

getc 函数能从键盘或者文件中获得单个字符。如果碰到 EOF，getc 函数会返回空字符串。

格式：

```
getc(FILEHANDLE)
getc FILEHANDLE
getc
```

示例 5.28

```
(The Script)
    # Getting only one character of input
    print "Answer y or n   ";
1   $answer=getc;      # Gets one character from stdin
2   $restofit=<>;      # What remains in the input buffer is
                       # assigned to $restofit
3   print "$answer\n";
4   print "The characters left in the input buffer were:
           $restofit\n";
(Output)
1   Answer  y or n yessirreebob <ENTER>
3   y
4   The characters left in the input buffer were: essirreebob
```

解释：

1. getc 函数从输入缓冲区中只读入一个字符，并存储到标量型变量 $answer 中。
2. 将输入缓冲区中剩余的字符存储到 $restofit 中，然后清空缓冲区。如果后面的程序需要读取输入，则不必再去从缓冲区中获取那些剩下的字符。
3. 打印由 getc 函数读入的字符串。
4. 打印存储在 $restofit 变量中的字符。

5.3.5　将输入内容赋予数组

当从文件句柄 STDIN 读取输入内容时，如果其上下文语境是数组型的，则读取到的每一行内容都将带着换行符作为独立的数组项来对待。用户可连续读取输入，直到碰见表示文件结束符（EOF）的 <Ctrl>-d（UNIX 中）或 <Ctrl>-z（Windows 中）。一般最好不要直接把输入内容赋给数组，因为这样做可能会占据大量内存，或者会碰到用户忘记按下 <Ctrl>-d 或 <Ctrl>-z 来终止输

入的情况。

示例 5.29

```
(The Script)
    # Assigning input to an array
1   print "Tell me everything about yourself.\n ";
2   @all = <STDIN>;
3   print "@all";
4   print "The number of elements in the array are: ",
           $#all + 1, ".\n";
5   print "The first element of the array is: $all[0]";

(Output)
1   Tell me everything about yourself.
2   OK. Let's see I was born before computers.
    I grew up in the 50s.
    I was in the hippie generation.
    I'm starting to get bored with talking about myself.
    <Ctrl>-d
3   OK. Let's see I was born before computers.
    I grew up in the 50s.
    I was in the hippie generation.
    I'm starting to get bored with talking about myself.
4   The number of elements in the array are: 4.
5   The first element of the array is:
    OK. Let's see I was born before computers.
```

解释：

1. 将字符串 Tell me everything about yourself 打印到 STDOUT。
2. 位于输入运算符 <> 之间的 STDIN 文件句柄负责读取输入行内容，直到碰见 EOF 文件结束符，或者碰到用户按下 <Ctrl>-d（UNIX 中）或 <Ctrl>-z（Windows 中）。每一个输入行与其末尾的换行符一起保存为数组 @all 中的元素。
3. 当用户按下 <Ctrl>-d 或 <Ctrl>-z 后，程序会把用户输入内容打印到屏幕上。
4. $# 结构负责获得数组中最后一个元素的下标或索引值。只要把 $#all 加上 1，就得到了该数组的容量，也就是需要读取的行数。
5. $all[0] 表示用户输入第一行数组里的第一个元素。其中，读取到的每一行都会成为数组中的一个元素。

5.3.6 将输入内容赋予散列

示例 5.30

```
(The Script)
    # Assign input to a hash
1   $course_number=101;
2   print "What is the name of course 101?";
3   chomp($course{$course_number} = <STDIN>);
4   print %course, "\n";

(Output)
2   What is the name of course 101? Linux Administration
4   101Linux Administration
```

解释：

1. 把标量型变量 $course_number 赋值为 101。
2. 将字符串 What is the name of course 101? 打印到 STDOUT。
3. 散列的名字是 %course。这里为该散列赋值，它的键是位于花括号中的 $course_number 变量值。chomp 函数负责从用户输入的值中删去换行符。
4. 打印新的数组。该数组含有一个键和相应的一个值。

5.4　数组函数

数组是可以扩大或者缩小的。Perl 提供了一系列数组函数，能够在数组的开头、中间或末尾进行元素的插入、删除操作。

5.4.1　chop 和 chomp 函数（用于列表）

chop 函数负责截去字符串的最后一个字符，并返回删掉的这个字符值。其用途一般是在把输入内容赋予标量型变量时删除其末尾的换行符。如果处理的是列表，则 chop 函数将删除数组中每个字符串的最后一个字符。

chomp 函数能够删除列表中每个带有换行符的元素的最后一个字符，并返回删掉的换行符数目。

格式：

```
chop(LIST)
chomp(LIST)
```

示例 5.31

```
(In the Script)
    # Chopping and chomping a list
1   @line=("red", "green", "orange");
2   chop(@line);    # Chops the last character off each
                    # string in the list
3   print "@line";
4   @line=( "red", "green", "orange");
5   chomp(@line);   # Chomps the newline off each string in the list
6   print "@line";

(Output)
3   re gree orang
6   red green orange
```

解释：

1. 把数组 @line 赋值为列表元素。
2. 对该数组进行 chop 处理。chop 函数会删除数组中每个元素的最后一个字符。
3. 打印得到的数组内容。
4. 把元素值赋予数组 @line。
5. chomp 函数会删除数组中每个元素末尾的换行符。该函数要比 chop 更加安全。
6. 如果数组元素末尾不含换行符，则不会删除它。

5.4.2 exists 函数

如果数组的下标（或散列表的键）已经定义，则 exists 函数会返回 true，否则就会返回 false。

格式：

```
exists $ARRAY[index];
```

示例 5.32

```
     #!/usr/bin/perl
1    @names = qw(Tom Raul Steve Jon);
2    print "Hello $names[1]\n", if exists $names[1];
3    print "Out of range!\n", if not exists $names[5];

(Output)
2    Hello Raul
3    Out of range!
```

解释：

1. 数组的名字是 @names。
2. 如果下标 1 已有定义，则 exists 函数会返回 true，并打印该字符串。
3. 如果下标 5 尚未定义（在本例中确实尚未定义），则打印字符串 Out of range!。

5.4.3 delete 函数

delete 函数用于从数组元素中删除指定的值，但它不会删除该元素本身，只是将它的值设定为“未定义”。

示例 5.33

```
(The Script)
# Removing an array element
1 @colors=("red","green","blue","yellow");
2  print "@colors\n";
3 delete $colors[1];  # green is removed
4 print "@colors\n";
5 print $colors[1],"\n";
6 $size=@colors;      # value is now undefined
7 print "The size of the array is $size.\n";

(Output)
2 red green blue yellow
4 red  blue yellow

7 The size of the array is 4.
```

5.4.4 grep 函数

grep 函数能够对数组（LIST）中的每个元素都求出表达式（EXP）的值，并据此返回另一个数组，该数组仅由上述表达式值为真的元素组成。如果其返回值是标量的话，则保存了表达式值为真的次数（即找到匹配样式的次数）。

格式：

```
grep(EXPR,LIST)
```

示例 5.34

```
(The Script)
    # Searching for patterns in a list
1   @list = (tomatoes, tomorrow, potatoes, phantom, Tommy);
2   $count = grep( /tom/i, @list);
    @items= grep( /tom/i, @list);
    print "Found items: @items\nNumber found: $count\n";

(Output)
4   Found items: tomatoes tomorrow phantom Tommy
    Number found: 4
```

解释：

1. 把数组 @line 赋值为列表元素。
2. grep 函数会搜索正则表达式 tom，其中 i 表示关闭大小写敏感选项。当把返回值赋予标量型变量时，其结果是正则表达式匹配成功的次数。
3. grep 函数会再次搜索正则表达式 tom，其中 i 表示关闭大小写敏感选项。当把返回值赋予数组型变量时，其结果是匹配成功的元素列表。

5.4.5　join 函数

join 函数能够把数组元素连接到单个字符串中，并借助指定的定界符划分各个数组元素。该函数功能正好与 split 函数（见“split 函数”一节）相反。用户可以在通过 split 函数将字符串分割为数组元素后再使用本函数。表达式 DELIMITER 是分隔数组元素的定界符的取值。LIST 则由数组元素构成。

格式：

```
join(DELIMITER, LIST)
```

示例 5.35

```
(The Script)
    # Joining each elements of a list with colons
1   $name="Joe Blow";
    $birth="11/12/86";
    $address="10 Main St.";
2   print join(":", $name, $birth, $address ), "\n";

(Output)
2   Joe Blow:11/12/86:10 Main St.
```

解释：

1. 把字符串赋值给标量。
2. join 函数借助冒号定界符连接三个标量，然后打印生成的字符串。

示例 5.36

```
(The Script)
    # Joining each element of a list with a newline
1   @names=('Dan','Dee','Scotty','Liz','Tom');
```

```
2    @names=join("\n", sort(@names));
3    print @names,"\n";

(Output)
3    Dan
     Dee
     Liz
     Scotty
     Tom
```

解释：

1. 把数组 @line 赋值为列表元素。
2. 在按照字母表顺序对列表进行排序后，再用 join 函数以换行符（\n）将每个数组元素连接起来。
3. 打印保存的列表内容，其中每个元素都占一行。

5.4.6 map 函数

map 函数能把数组中的每个元素值映射到表达式或者块中，并返回另一个数组。这里通过示例来介绍更好理解。

格式：

```
map EXPR, LIST;
map {BLOCK} LIST;
```

示例 5.37

```
(The Script)
     # Mapping a list to an expression
1    @list=(0x53,0x77,0x65,0x64,0x65,0x6e,012);
2    @words = map chr, @list;
3    print @words;
4    @n = (2, 4, 6, 8);
5    @n = map $_ * 2 + 6, @n;
6    print "@n\n";

(Output)
3    Sweden
6    10 14 18 22
```

解释：

1. 数组 @list 由 6 个十六进制整数和 1 个八进制整数组成。
2. map 函数将 @list 数组中的每一项映射为其对应的 chr（字符）值，并返回新的列表。
3. 打印新数组内容。使用 chr 函数把每个数字值转换为与该 ASCII 值对应的字符。
4. 数组 @n 由一组整数组成。
5. map 函数能够对 @n 数组中的每个元素分别匹配正则表达，并将匹配得到的新数组返回到 @n 中。
6. 打印映射的结果。

示例 5.38

```
(The Script)
     # Map using a block
```

```
1   open(FH, "datebook.master") or die;
2   @lines=<FH>;
3   @fields = map { split(":") } @lines;
4   foreach $field (@fields){
5       print $field,"\n";
    }
(Output)
5   Sir Lancelot
    837-835-8257
    474 Camelot Boulevard, Bath, WY 28356
    5/13/69
    24500

    Tommy Savage
    408-724-0140
    1222 Oxbow Court, Sunnyvale, CA 94087
    5/19/66
    34200

    Yukio Takeshida
    387-827-1095
    13 Uno Lane, Asheville, NC 23556
    7/1/29
    57000

    Vinh Tranh
    438-910-7449
    8235 Maple Street, Wilmington, VT 29085
    9/23/63
    68900
```

解释：

1. 打开 datebook.master 文件，读取 FH 文件句柄。该文件每一行内容都是以逗号隔开的字段，并以换行符结尾。
2. 读取文件内容，并将其赋予数组 @lines。文件的每一行内容都将成为数组的一个元素。
3. map 函数设定为使用块（block）格式。split 函数将把数组元素内容从分号处隔开，从而得到新的列表，其列表项成为数组的元素。
4. 借助 foreach 循环遍历整个数组，依次将每个元素值赋给 $field。
5. 显示映射的效果。在映射前，行的内容是：Sir Lancelot: 837-835-8257:474 Camelot Boulevard, Bath, WY 28356:5/13/69:24500

5.4.7　pack 和 unpack 函数

pack 和 unpack 函数用途很多。它们可用于把列表内容压缩成二进制结构，或者将压缩的值展开成列表。当用于文件时，这两个函数能用于创建解码文件、关系型数据库和二进制文件。

pack 函数能够将列表转换为可存于内存中的标量值。其中，TEMPLATE 负责指定字符的类型以及如何格式化字符。例如，字符串 c4 或 cccc 表明把列表压缩为 4 个无符号的字符；而 a14 则表明将列表压缩为长度是 14 个字节的 ASCII 字符串，并用空白字符补足多余的字符。unpack 函数能把二进制格式的字符串转换为列表，并将字符串变回 Perl 格式。

表 5-2 pack 和 unpack 所用模板类型和值

模　板	描　述
a	ASCII 字符串（以空值补全）
A	ASCII 字符串（以空格补全）
b	比特字符串（按从低位到高位的顺序）
B	比特字符串（按从高位到低位的顺序）
c	带符号的字符值
C	无符号的字符值
d	本机格式支持的双精度浮点数
f	本机格式支持的单精度浮点数
h	十六进制字符串（从低位到高位的 4 位表示法）
H	十六进制字符串（从高位到低位的 4 位表示法）
i	带符号整数
I	无符号整数
l	带符号长整数
L	无符号长整数
n	以“network”（高位在前）为顺序的短整数
N	以“network”（高位在前）为顺序的长整数
p	指向以空值结束的字符串的指针
P	指向结构（定长字符串）的指针
q	带符号的 64 位值
Q	无符号的 64 位值
s	带符号的短值（16 位）
S	无符号的短值（16 位）
u	未编码的字符串
v	以“VAX”（低位在前）为顺序的短值
V	以“VAX”（低位在前）为顺序的长值
w	128 为基数的 BER 压缩无符号整数，高位在前
x	空字节
X	字节的备份值
@	填入到绝对位置的空值

5.4.8 pop 函数

pop 函数能弹出数组的最后一个元素，并返回该元素内容。此后，该数组的长度将减 1。

格式：

```
pop(ARRAY)
pop ARRAY
```

示例 5.39

```
(In Script)
    # Removing an element from the end of a list
1   @names=("Bob", "Dan", "Tom", "Guy");
2   print "@names\n";
3   $got = pop(@names);   # Pops off last element of the array
4   print "$got\n";
5   print "@names\n";

(Output)
2   Bob Dan Tom Guy
4   Guy
5   Bob Dan Tom
```

解释：

1. 把数组 @names 赋值为列表元素。
2. 打印该数组。
3. pop 函数将移除该数组的最后一个元素，并返回该弹出项内容。
4. 标量 $got 中含有弹出项 Guy。
5. 打印新的数组。

5.4.9　push 函数

push 函数能把一个数值追加到数组末尾，同时增大该数组的长度。

格式：

```
push(ARRAY, LIST)
```

示例 5.40

```
(In Script)
    # Adding elements to the end of a list
1   @names=("Bob", "Dan", "Tom", "Guy");
2   push(@names, "Jim", "Joseph", "Archie");
3   print "@names \n";

(Output)
2   Bob Dan Tom Guy Jim Joseph Archie
```

解释：

1. 把数组 @names 赋值为列表元素。
2. push 函数将把 3 个元素添加到数组的尾部。
3. 新数组中将含有刚添加的这 3 个元素。

5.4.10　shift 函数

shift 函数能移除并返回数组的第一个元素，同时把数组长度减去 1。如果省略掉数组名参数 ARRAY，则会移除 ARGV 数组的首元素；如果是在子例程中的话，则移除 @_ 数组。

格式：

```
shift(ARRAY)
shift ARRAY
shift
```

示例 5.41

```
(In Script)
    # Removing elements from front of a list
1   @names=("Bob", "Dan", "Tom", "Guy");
2   $ret  = shift @names;
3   print "@names\n";
4   print "The item shifted is $ret.\n";
(Output)
3   Dan Tom Guy
4   The item shifted is Bob.
```

解释：

1. 把数组 @names 赋值为列表元素。
2. shift 函数负责移除数组的第一个元素，并将该元素值返回到标量 $ret 中。该元素值是 Bob。
3. 新数组将减少一个元素。

5.4.11　splice 函数

splice 函数能移除并替换数组中的指定元素。其中，参数 OFFSET 指定需要移除的元素起始位置，LENGTH 则指定从 OFFSET 位置开始需要移动的元素数目。LIST 则指明用户替换旧元素的新元素内容。

格式：

```
splice(ARRAY, OFFSET, LENGTH, LIST)
splice(ARRAY, OFFSET, LENGTH)
splice(ARRAY, OFFSET)
```

示例 5.42

```
(The Script)
    # Splicing out elements of a list
1   @colors=("red", "green", "purple", "blue", "brown");
2   print "The original array is @colors\n";
3   @discarded = splice(@colors, 2, 2);
4   print "The elements removed after the splice are: @discarded.\n";
5   print "The spliced array is now @colors.\n";

(Output)
2   The original array is red green purple blue brown
4   The elements removed after the splice are: purple blue.
5   The spliced array is now red green brown.
```

解释：

1. 创建一个含有 5 种颜色值的数组。
2. 打印原数组内容。

3. splice 函数将移除从偏移量 2（偏移量是从 0 开始计算的）开始的两个元素：purple 和 blue，并将删除的元素返回到另一个名为 @discarded 的数组。
4. splice 函数移除元素 purple 和 blue，并将其返回到 @discarded。其内容从元素 $colors[2] 开始，长度是 2 个元素。
5. 拼接 @colors 数组。删除其中的元素 purple 和 blue。

示例 5.43

```
(The Script)
    # Splicing and replacing elements of a list
1   @colors=("red", "green", "purple", "blue", "brown");
2   print "The original array is @colors\n";
3   @lostcolors=splice(@colors, 2, 3, "yellow", "orange");
4   print "The removed items are @lostcolors\n";
5   print "The spliced array is now @colors\n";

(Output)
2   The original array is red green purple blue brown
4   The removed items are purple blue brown
5   The spliced array is now red green yellow orange
```

解释：

1. 创建一个含有 5 种颜色值的数组。
2. 打印原数组内容。
3. splice 函数将移除从偏移量 2（偏移量是从 0 开始计算的）开始的两个元素，并删除接下来的 3 个元素。删除的元素(purple、blue 和 brown)存储在 @lostcolors 中。然后用颜色 yellow 和 orange 替换被删除的颜色。
4. 被删除的颜色值保存在 @lostcolors 数组中，打印该数组内容。
5. 打印拼接后的数组内容。

5.4.12　split 函数

split 函数能够基于一些定界符（默认为空白字符）拆分字符串（EXPR），并返回一个数组。其第一个参数表示定界符，第二个参数指定待拆分的字符串。Perl 的 split 函数可以在处理文本时用于创建字段，正如使用 awk 一样。如果没有指明待拆分的字符串，则该函数会默认拆分 $_ 字符串。

DELIMITER 语句负责匹配那些用于分隔字段的定界符。如果在程序中省略掉 DELIMITER 的话，即表示定界符是空白字符（即空格、制表符、换行符等）。如果 DELIMITER 与定界符不匹配的话，split 函数便会返回原始字符串。用户可使用正则表达式元字符 [] 来指定多个定界符。譬如，[+\t:] 表示 0 到多个空格符或者制表符或分号。

LIMIT 负责指定需要拆分的字段数目。如果存在多个 LIMIT 字段，则其他字段都是最后一个字段的一部分。如果在程序中省略了 LIMIT 的话，split 函数会使用其自带的 LIMIT，它比 EXPR 的字段数目多 1（有关自动拆分的模式请参阅附录 A 中的 -a 开关）。

格式：

```
split("DELIMITER",EXPR,LIMIT)
```

```
split(/DELIMITER/,EXPR,LIMIT)
split(/DELIMITER/,EXPR)
split("DELIMITER",EXPR)
split(/DELIMITER/)
split
```

示例 5.44

```
(The Script)
    # Splitting a scalar on whitespace and creating a list
1   $line="a b c d e";
2   @letter=split(' ',$line);
3   print "The first letter is $letter[0]\n";
4   print "The second letter is $letter[1]\n";

(Output)
3   The first letter is a
4   The second letter is b
```

解释：

1. 把字符串 a b c d e 赋值给标量型变量 $line。
2. $line（标量）中的值是单字母型字符串。split 函数能以空白字符为定界符拆分该字符串，并把数组 @letter 赋值为单个字母 a、b、c、d 和 e。使用单引号作为定界符时，其情况与使用正则表达式 // 是不同的。在以空白字符拆分时，' ' 的作用类似于 awk，它会忽略开头的空白字符。而正则表达式 // 则会保留开头的空白字符，并创建与这些空白字符一样多的空白字段。
3. 打印 @letter 数组的第一个元素。
4. 打印 @letter 数组的第二个元素。

示例 5.45

```
(The Script)
    # Splitting up $_
1   while(<DATA>){
2       @line=split(":");      # or split (":", $_);
3       print "$line[0]\n";
    }
_ _DATA_ _
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ 23875:6/18/68:23400
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700
Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX 83745:10/1/35:58900
Fred Fardbarkle:674-843-1385:20 Park Lane, Duluth, MN 23850:4/12/23:78900

    (Output)
    Betty Boop
    Igor Chevsky
    Norma Corder
    Jennifer Cowan
    Fred Fardbarkle
```

解释：

1. $_ 变量能够保存 DATA 文件句柄的每一行内容；需要处理的数据位于 _DATA_ 行之下。其每一行内容都将传给 $_ 变量，后者也是 split 函数的默认行。
2. split 函数使用 $_ 来拆分各行内容，利用冒号 : 作为定界符，并返回结果到数组 @line。

3. 打印 @line 数组的第一个元素 line[0]。

示例 5.46

```
 (The Script)
      # Splitting up $_ and creating an unnamed list
      while(<DATA>){
  1   ($name,$phone,$address,$bd,$sal)=split(":");
  2   print "$name\t $phone\n" ;
      }
_ _DATA_ _
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
Igor Chevsky:385-375-8395:3567 Populus Place, Caldwell, NJ 23875:6/18/68:23400
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700
Jennifer Cowan:548-834-2348:583 Laurel Ave., Kingsville, TX 83745:10/1/35:58900
Fred Fardbarkle:674-843-1385:20 Park Lane, Duluth, MN 23850:4/12/23:78900

  (Output)
  2   Betty Boop  245-836-8357
      Igor Chevsky  385-375-8395
      Norma Corder  397-857-2735
      Jennifer Cowan  548-834-2348
      Fred Fardbarkle  674-843-1385
```

解释：

1. Perl 会逐行遍历 DATA 文件句柄，并将文件的每一行内容存入 $_ 变量中。split 函数则负责拆分每一行内容，并以冒号 : 为定界符。
2. 该数组含有 5 个标量：$name、$phone、$address、$db 和 $sal。打印其中 $name、$phone 的值。

示例 5.47

```
(The Script)
    # Many ways to split a scalar to create a list
1   $string= "Joe Blow:11/12/86:10 Main St.:Boston, MA:02530";
2   @line=split(":", $string);    # The string delimiter is a colon
3   print @line,"\n";
4   print "The guy's name is $line[0].\n";
5   print "The birthday is $line[1].\n\n";

6   @str=split(":", $string, 2);
7   print $str[0],"\n";  # The first element of the array
8   print $str[1],"\n";  # The rest of the array because limit is 2
9   print $str[2],"\n";  # Nothing is printed

10  @str=split(":", $string);  # Limit not stated will be one more
                               # than total number of fields
11  print $str[0],"\n";
12  print $str[1],"\n";
13  print $str[2],"\n";
14  print $str[3],"\n";
15  print $str[4],"\n";
16  print $str[5],"\n";

17  ( $name, $birth, $address )=split(":", $string);
              # Limit is implicitly 4, one more than
```

```
                # the number of fields specified
18  print $name , "\n";
19  print $birth,"\n";
20  print $address,"\n";

(Output)
3   Joe Blow11/12/8610 Main St.Boston, MA02530
4   The guy's name is Joe Blow.
5   The birthday is 11/12/86.

7   Joe Blow
8   11/12/86:10 Main St.:Boston, MA:02530
9
11  Joe Blow
12  11/12/86
13  10 Main St.
14  Boston, MA
15  02530
16
18  Joe Blow
19  11/12/86
20  10 Main St.
```

解释：

1. 以逗号为定界符拆分标量 $string。
2. 定界符是逗号，其限制值为 2。
3. 如果没有说明的话，LIMIT 的默认值是字段总数加 1。
4. LIMIT 的隐含值是 4，比规定的字段多 1。

5.4.13 sort 函数

sort 函数能返回经过排序后的数组。如果没有指定 SUBROUTINE，则该函数将按照字符串先后顺序进行排序。如果程序中指定了 SUBROUTINE，则 sort 函数的第一个参数将是排序子例程的名称，其后第二个参数则是一组待排序的值。如果在程序中使用了字符串的 cmp 运算符，则这些值将按照字母表顺序（即 ASCII 表中的顺序）进行排序；如果使用的是 <=> 运算符（又称作“space ship”运算符）的话，则以数字顺序对这些值进行排序。根据比较结果，排序子例程会返回大于、小于或等于 0 的整数值。值是通过引用方式传送给子例程的，该值由特殊变量 $a 和 $b 表示，而不是一般的 @_ 数组。（更多介绍请参阅第 11 章“子例程”）。请务必不要修改变量 $a 或 $b，因为它们存有待排序的内容。

如果读者需要让 Perl 根据某种特定语言对数据进行排序，就应当在程序中包含一个 use locale 编译标识符。有关这方面的详细信息和编程步骤，请读者参考：http://search.cpan.org/~nwclark/Perl-5.8.8/pod/perllocale.pod。

格式：

```
sort(SUBROUTINE LIST)
sort(LIST)
sort SUBROUTINE LIST
sort LIST
```

示例 5.48

```
(The Script)
# Simple alphabetic sort
1 @list=("dog","cat", "bird","snake" );
  print "Original list: @list\n";
2 @sorted = sort(@list);
3  print "Ascii sort: @sorted\n";

 # Reversed alphabetic sort
4  @sorted = reverse(sort(@list));
   print "Reversed Ascii sort: @sorted\n";

(Output)
Original list: dog cat bird snake
Ascii sort: bird cat dog snake
Reversed Ascii sort: snake dog cat bird
```

解释：

1. @string 数组中含有待排序项目的列表。
2. sort 函数负责对项目执行字符串（ASCII）排序。排序得到的值必须赋值到另一个数组或原数组中。sort 函数不会改变原数组的内容。
3. 打印排序后的数组内容。
4. 首先按照字母表顺序对列表进行排序，然后翻转该列表。

1. 使用子例程完成 ASCII 型与数值型排序

示例 5.49

```
(The Script)
1 @list=("dog","cat", "bird","snake" );
  print "Original list: @list\n";
   # ASCII sort using a subroutine
2  sub asc_sort{
3          $a cmp $b;  # Sort ascending order
   }
4  @sorted_list=sort asc_sort(@list);
   print "Ascii sort: @sorted_list\n";

   # Numeric sort using subroutine
5  sub numeric_sort {
       $a <=> $b ;
   }  # $a and $b are compared numerically

6  @number_sort=sort numeric_sort 10, 0, 5, 9.5, 10, 1000;
        print "Numeric sort: @number_sort.\n";
(Output)
Original list: dog cat bird snake
Ascii sort: bird cat dog snake
Numeric sort: 0 5 9.5 10 10 1000.
```

解释：

1. @list 数组中含有一组待排序的数据项。
2. 为子例程 asc_sort() 提供一组待排序的字符串。

3. 特殊变量 $a 和 $b 会按升序比较待排序的数据项。如果把 $a 和 $b 的顺序反过来（譬如 $b cmp $a），则上述排序操作将按照降序进行。在比较字符串时，需要用到 cmp 操作符。
4. sort 函数会把一个列表发送给用户定义的子例程 asc_sort()，并由后者完成排序操作。排序后得到的列表将返回并存入到 @sorted_list 数组中。
5. 本行提供了一个用户定义的子例程，其名称是 numeric_sort()。特殊变量 $a 和 $b 会按数字升序比较待排序的数据项。如果把 $a 和 $b 的顺序反过来（譬如 $b cmp $a），则上述排序操作将按照降序进行。在比较数值时，需要用到 <=> 操作符。
6. sort 函数会把一个列表发送给用户定义的子例程 numeric_sort()，获得排序后的数字列表，并将结果保存到 @number_sort 数组中。

2. 使用内联 (Inline) 函数对数值列表进行排序

示例 5.50

```
(The Script)
    # Sorting numbers with an unamed subroutine
1   @sorted_numbers= sort {$a <=> $b} (3,4,1,2);
2   print "The sorted numbers are: @sorted_numbers", ".\n";

(Output)
2   The sorted numbers are: 1 2 3 4.
```

解释：

1. 为 sort 函数提供一个无名的子例程，又称作内联函数，用于对通过参数传入的数值型列表进行排序。<=> 运算符和特殊变量 $a、$b 一起实现对数字的排序。排序得到的数值型列表则保存到 @sorted_numbers 数组中。
2. 打印排序得到的数组内容。

5.4.14 reverse 函数

reverse 函数负责翻转数组中元素的排列顺序，使得原本降序排列的数组元素变成升序，反之亦然。

格式：

```
reverse(LIST)
reverse LIST
```

示例 5.51

```
(In Script)
    # Reversing the elements of an array
1   @names=("Bob", "Dan", "Tom", "Guy");
2   print "@names \n";
3   @reversed=reverse(@names),"\n";
4   print "@reversed\n";

(Output)
2   Bob Dan Tom Guy
4   Guy Tom Dan Bob
```

解释：

1. 将列表赋值给数组 @names。
2. 打印原数组内容。
3. reverse 函数将翻转数组中元素的顺序，并返回得到的新数组。原数组 @names 将没有变化。新得到的数组会保存到 @reversed 中。
4. 打印翻转后的数组内容。

5.4.15　unshift 函数

unshift 函数能把 LIST 追加到数组的起始位置。

格式：

```
unshift(ARRAY, LIST)
```

示例 5.52

```
(In Script)
    # Putting new elements at the front of a list
    @names=("Jody", "Bert", "Tom") ;
    unshift(@names, "Liz", "Daniel");
    print "@names\n";

(Output)
    Liz Daniel Jody Bert Tom
```

解释：

1. 向数组 @names 赋 3 个值：Jody、Bert 和 Tom。
2. unshift 函数会在数组前面追加 2 个元素 Liz 与 Daniel。
3. 打印 @names 数组。

5.5　散列（关联数组）函数

5.5.1　keys 函数

keys 函数将返回一个数组，该数组的所有元素都是指定散列表（Hash）的键（请参考“values 函数”和“each 函数”相关内容）。

格式：

```
keys(ASSOC_ARRAY)
keys ASSOC_ARRAY
```

示例 5.53

```
(In Script)
    # The keys function returns the keys of a hash
    %weekday= (
                 '1'=>'Monday',
```

```
                 '2'=>'Tuesday',
                 '3'=>'Wednesday',
                 '4'=>'Thursday',
                 '5'=>'Friday',
                 '6'=>'Saturday',
                 '7'=>'Sunday',
                );
2   foreach $key ( keys(%weekday) ){print "$key ";}
    print "\n";
3   foreach $key ( sort keys(%weekday) ){print "$key" ;}
    print "\n";

(Output)
2   7 1 2 3 4 5 6
3   1 2 3 4 5 6 7
```

解释：

1. 为散列变量 %weekday 赋予键和值。
2. 针对散列变量 %weekday 中的每一项，调用 keys 函数获得它的键。然后把键的内容赋值给标量 $key，并以随机顺序打印它们。
3. 最后对这些键进行排序和打印。

5.5.2 values 函数

values 函数能以随机顺序返回含有一个散列表中所有值的数组。

格式：

```
values(ASSOC_ARRAY)
values ASSOC_ARRAY
```

示例 5.54

```
(In Script)
    # The values function returns the values in a hash
1   %weekday= (
                 '1'=>'Monday',
                 '2'=>'Tuesday',
                 '3'=>'Wednesday',
                 '4'=>'Thursday',
                 '5'=>'Friday',
                 '6'=>'Saturday',
                 '7'=>'Sunday',
                );
2   foreach $value ( values(%weekday)){print "$value";}
    print "\n";

(Output)
2   Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

解释：

1. 为散列变量 %weekday 赋予键和值。
2. 针对散列变量 %weekday 中的每一项，调用 values 函数得到它的值。然后，把值赋给标量 $value，并打印到 STDOUT。

5.5.3 each 函数

each 函数能以随机顺序返回一个双元素数组，该数组的一个元素是散列表项的键，另一个元素则是该键所对应的值。

格式：

```
each(ASSOC_ARRAY)
```

示例 5.55

```
(In Script)
#! /usr/bin/perl
# The each function retrieves both keys and values from a hash
1   %weekday=(
              'Mon' => 'Monday',
              'Tue' => 'Tuesday',
              'Wed' => 'Wednesday',
              'Thu' => 'Thursday',
              'Fri' => 'Friday',
              'Sat' => 'Saturday',
              'Sun' => 'Sunday',
             );
2  while(($key,$value)=each(%weekday)){
3      print "$key = $value\n";
   }

(Output)
3   Sat = Saturday
    Fri = Friday
    Sun = Sunday
    Thu = Thursday
    Wed = Wednesday
    Tue = Tuesday
    Mon = Monday
```

解释：

1. 为散列变量 %weekday 赋予键和值。
2. each 函数返回散列变量 %weekday 中的每个键及其对应值。然后分别将这些键和值赋给标量 $key 与 $value。
3. 以无序方式打印这些键和值。

5.5.4 对散列进行排序

如果要对一个散列进行排序，读者可以像前面对待数组一样，借助内建的 sort() 命令方便地按字母表顺序对键（key）进行排列。但也有可能需要按数字顺序排序，或者根据散列值来排序。为了满足这些需求，读者还需要自行定义一个子例程，用于比较键或者值的大小。（详见第 11 章）。内建的 sort() 函数将调用该子例程，并传给它一组待比较的键或者值。根据所用运算符的不同，该子例程可以按 ASCII（字母表）顺序，亦可按数字大小进行比较。其中，“cmp” 运算符用于比较字符串，而 “<=>” 运算符则负责比较数字。全局标量 $a 和 $b 负责在进行比较时为子例程保存比较的值，并且这两个标量名是固定不变的。

按键升序排列散列。对散列的键按 ASCII 或字母表顺序进行排列，是一项相对而言较为容易的工作。只需向 sort() 函数提供一组待排序的键，便可按升序排列它们了。这里需要使用一个 foreach 循环来逐个访问散列中的所有键。详见示例 5.56。

示例 5.56

```
(In Script)
1   %wins = (
        "Portland Panthers"   => 10,
        "Sunnyvale Sluggers"  => 12,
        "Chico Wildcats"      => 5,
        "Stevensville Tigers" => 6,
        "Lewiston Blazers"    => 11,
        "Danville Terriors"   => 8,
    );
    print "\n\tSort Teams in Ascending Order:\n\n";
2   foreach $key( sort(keys %wins)) {
3       printf "\t% -20s%5d\n", $key, $wins{$key};
    }
(Output)
Sort Teams in Ascending  Order:

        Chico Wildcats          5
        Danville Terriors       8
        Lewiston Blazers       11
        Portland Panthers      10
        Stevensville Tigers     6
        Sunnyvale Sluggers     12
```

解释：

1. 向散列变量 %wins 赋予一组键 / 值对。
2. 使用 foreach 循环遍历散列中的每个元素。foreach 循环将从 sort() 函数得到输出，该输出是一组按字母表顺序经过排序的列表。而 sort 函数则从内建的 keys 函数得到输入，因为 keys 函数能返回一个散列中所有键的列表。
3. printf() 函数用于格式化并打印各个键，以及排列后的值。

按键反序排列散列。如需根据散列的键按照字母表顺序进行反序排列，只需为前面的示例添加内建的 reverse() 函数便可。在完成反序排列后，再通过一个 foreach 循环逐个获取散列的每一个键。

示例 5.57

```
1  %wins = (
        "Portland Panthers"   => 10,
        "Sunnyvale Sluggers"  => 12,
        "Chico Wildcats"      => 5,
        "Stevensville Tigers" => 6,
        "Lewiston Blazers"    => 11,
        "Danville Terriors"   => 8,
   );
    print "\n\tSort Teams in Descending/Reverse Order:\n\n";
2   foreach $key (reverse sort(keys %wins)) {
3      printf "\t% -20s%5d\n", $key, $wins{$key};
```

```
    }

(Output)
Sort Teams in Descending/Reverse Order:

        Sunnyvale Sluggers      12
        Stevensville Tigers      6
        Portland Panthers       10
        Lewiston Blazers        11
        Danville Terriors        8
        Chico Wildcats           5
```

解释：

1. 向散列变量 %wins 赋予一组键 / 值对。
2. 使用 foreach 循环遍历散列中的每个元素。reverse() 函数负责从 sort() 函数获得排列后的列表数据，并完成反转处理。然后，foreach 函数使用翻转后的列表从散列变量 %wins 中解析出每个键和值。
3. printf() 函数用于格式化并打印各个键，以及排列后的值。

以数值方式按键排列散列。如需根据键的数值大小对散列进行排序，则需要使用用户自定义的子例程。在使用各种运算符执行比较时，标量 $a 和 $b 负责为子例程保存待比较的值。其中，“<=>”运算符负责比较数字“cmp”运算符则用于比较字符串。sort() 函数将把一组键传送给用户自定义的子例程，然后返回排序后的列表结果。

示例 5.58

```
1  sub desc_sort_subject {
2       $b <=> $a;          # Numeric sort descending
    }
3   sub asc_sort_subject{
4       $a <=> $b;          # Numeric sort ascending
    }
5    %courses = (
        "101" => "Intro to Computer Science",
        "221" => "Linguistics",
        "300" => "Astronomy",
        "102" => "Perl",
        "103" => "PHP",
        "200" => "Language arts",
    );
    print "\n\tCourses in Ascending Numeric Order:\n";
6    foreach $key (sort asc_sort_subject(keys(%courses))) {
7       printf "\t%-5d%s\n", $key, $courses{"$key"};
    }
    print "\n\tCourses in Descending Numeric Order:\n";
    foreach $key (sort desc_sort_subject(keys(%courses))) {
      printf "\t%-5d%s\n", $key, $courses{"$key"};
    }
(Output)
 Courses in Ascending Numeric Order:
        101  Intro to Computer Science
        102  Perl
        103  PHP
```

```
        200  Language arts
        221  Linguistics
        300  Astronomy

 Courses in Descending Numeric Order:
         300  Astronomy
         221  Linguistics
         200  Language arts
         103  PHP
         102  Perl
         101  Intro to Computer Science
```

解释：

1. 本行含有一个用户自定义的子例程 desc_sort_subject()。当在 sort() 函数中指定该子例程名时，该函数便会比较传进来的各个键，并按照数值大小对它们进行排序。
2. 标量 $a 和 $b 负责比较来自散列 %courses 的键值。<=> 是一个数值比较表达式，它会把每个键当作数值来进行排列。在前面示例中，我们还曾按照字母表顺序对键值进行排列。由于 $b 位于 $a 之前，因此上述排序操作是降序的。
3. 这里又是一个用户自定义的子例程，名叫 asc_sort_subject()。该函数和前面第一行的函数一模一样，惟一的差别在于它将按照数值升序排列散列键，而不是降序。
4. 在这个函数中，特殊变量 $a 和 $b 顺序是反过来的，因此比较后的排序结果将是升序的。
5. 向散列变量 %courses 赋予多个键 / 值对。
6. 使用 foreach 循环逐个遍历散列中的每个元素。其输入则来自于 sort() 函数的输出。
7. printf() 函数用于格式化并打印各个键，以及排列后的值。

以数值方式按值升序排列散列。如需根据值的大小对散列进行升序排列，则需要使用用户自定义的子例程。标量 $a 和 $b 负责比较来自散列的值。如果 $a 位于比较运算符的左侧，则该排序过程将是升序的；反之，如果 $b 位于左侧，则排序过程将是降序的。<=> 运算符将以数值方式比较左右两个参数。

示例 5.59

```
(In Script)
1  sub asc_sort_wins {
2       $wins{$a} <=> $wins{$b};
   }


3    %wins = (
        "Portland Panthers"   => 10,
        "Sunnyvale Sluggers"  => 12,
        "Chico Wildcats"      => 5,
        "Stevensville Tigers" => 6,
        "Lewiston Blazers"    => 11,
        "Danville Terriors"   => 8,
   );
   print "\n\tWins in Ascending Numeric Order:\n\n";
4   foreach $key (sort asc_sort_wins(keys(%wins))) {
5       printf "\t% -20s%5d\n", $key, $wins{$key};
    }
```

```
(Output)

Wins in Ascending Numeric Order:

        Chico Wildcats           5
        Stevensville Tigers      6
        Danville Terriors        8
        Portland Panthers       10
        Lewiston Blazers        11
        Sunnyvale Sluggers      12
```

解释：

1. 这是一个用户自定义的子例程，其名字是 asc_sort_wins()。当在 sort() 函数中指定该子例程名时，该函数便会比较传进来的各个散列值，然后以数值方式按值的大小进行排序。
2. 特殊 Perl 变量 $a 和 $b 负责比较来自散列 %wins 的值。<=> 运算符将以数值方式比较每个待排序的值。如需对字符串进行比较，则应使用“cmp”运算符。
3. 向散列变量 %wins 赋予多个键 / 值对。
4. 使用 foreach 循环逐个遍历散列中的每个元素。其输入则来自于 sort() 函数的输出。
5. printf() 函数用于格式化并打印各个键，以及排列后的值。

以数值方式按值降序排列散列。与前面的示例一样，如需根据值的大小对散列进行降序排列，则需要使用用户自定义的子例程。不过这一次 $b 变量应当位于数值比较运算符 <=> 的左侧，而 $a 变量应当位于其右侧。这样就能让 sort() 函数以降序进行排列。

示例 5.60

```
    (In Script)
   # Sorting a Hash by Value in Descending Order

1  sub desc_sort_wins {
2       $wins{$b} <=> $wins{$a};  # Reverse $a and $b
   }

3  %wins = (
        "Portland Panthers"   => 10,
        "Sunnyvale Sluggers"  => 12,
        "Chico Wildcats"      => 5,
        "Stevensville Tigers" => 6,
        "Lewiston Blazers"    => 11,
        "Danville Terriors"   => 8,
   );
   print "\n\tWins in Descending Numeric Order:\n\n";
4  foreach $key (sort desc_sort_wins(keys(%wins))) {
5      printf "\t% -20s%5d\n", $key, $wins{$key};
   }
(Output)

Wins in Descending Numeric Order:

        Sunnyvale Sluggers      12
        Lewiston Blazers        11
```

```
          Portland Panthers       10
          Danville Terriors        8
          Stevensville Tigers      6
          Chico Wildcats           5
```

解释：

1. 这是一个用户自定义的子例程，其名字是 desc_sort_wins()。当在 sort() 函数中指定该子例程名时，该函数便会比较传进来的各个散列值，然后以数值方式按值的大小进行排序。只不过这次是按照降序进行排列的。
2. 特殊 Perl 变量 $a 和 $b 负责比较来自散列 %wins 的值。$a 和 $b 的相对位置表明了排列顺序究竟是升序还是降序。如果 $a 位于数值比较运算符 <=> 的左侧，则该排序过程将是升序的；反之，如果 $b 位于数值比较运算符 <=> 的左侧，则该排序过程则是降序的。如需对字符串进行比较，则应使用“cmp”运算符。
3. 向散列变量 %wins 赋予多个键 / 值对。
4. 使用 foreach 循环逐个遍历散列中的每个元素。其输入则来自于 sort() 函数的输出。
5. printf() 函数用于格式化并打印各个键，以及排列后的值。

5.5.5　delete 函数

delete 函数负责从散列中删除指定值。如果成功就返回删掉的这个值。⊖

格式：

```
delete $ASSOC_ARRAY{KEY}
```

示例 5.61

```
(In Script)
    #!/usr/bin/perl
1   %employees=(
                 "Nightwatchman" => "Joe Blow",
                 "Janitor" => "Teddy Plunger",
                 "Clerk" => "Sally Olivetti",
               );
2   $layoff=delete $employees{"Janitor"};
    print "We had to let $layoff go.\n";
    print "Our remaining staff includes: ";
    print "\n";
    while(($key, $value)=each(%employees)){
       print "$key: $value\n";
    }

(Output)
We had to let Teddy Plunger go.
Our remaining staff includes:
Nightwatchman: Joe Blow
Clerk: Sally Olivetti
```

解释：

1. 用 3 个键 / 值对定义一个散列型变量。

⊖ 如果删除了 %ENV 中的值，则会影响系统环境变量（详见“%ENV 散列”一节）。

2. 根据指定的键，delete 函数将从散列中删除一个相应的元素。在此过程中，该元素的键和值都会删除。
3. 删除并返回与键 Janitor 相关联的散列值。然后把返回值 Teddy Plungers 赋予标量 $layoff。

5.5.6 exists 函数

如果散列的键（或数组下标）已经定义，则 exists 函数返回 true，否则返回 false。

格式：

```
exists $ASSOC_ARRAY{KEY}
```

示例 5.62

```
    #!/usr/bin/perl
1   %employees=( "Nightwatchman" => "Joe Blow",
                 "Janitor" => "Teddy Plunger",
                 "Clerk" => "Sally Olivetti",
               );
2   print "The Nightwatchman exists.\n" if exists
          $employees{"Nightwatchman"};
3   print "The Clerk exists.\n" if exists $employees{"Clerk"};
4   print "The Boss does not exist.\n" if not exists
$employees{"Boss"};

(Output)
2   The Nightwatchman exists.
3   The Clerk exists.
4   The Boss does not exist.
```

解释：

1. 用 3 个键 / 值对定义一个散列型变量。
2. 如果已经定义了键“Nightwatchman”，exists 函数会返回 true。
3. 如果定义了键“Clerk”，exists 函数会返回 true。
4. 如果尚未定义键“Clerk”，exists 函数将返回相反的结果。

5.6 有关散列的更多内容

5.6.1 从文件载入散列

示例 5.63

```
(The Database)
    1 Steve Blenheim
    2 Betty Boop
    3 Igor Chevsky
    4 Norma Cord
    5 Jon DeLoach
    6 Karen Evich
```

```
(The Script)
    #!/usr/bin/perl
    # Loading a Hash from a file.
1   open(NAMES,"emp.names") || die "Can't open emp.names: $!\n";
2   while(<NAMES>){
3        ( $num, $name )= split(' ', $_, 2);
4        $realid{$num} = $name;
    }
5   close NAMES;

6   while(1){
7      print "Please choose a number from the list of names? ";
8      chomp($num=<STDIN>);
9      last unless $num;
10     print $realid{$num},"\n";
    }

(Output)
7   Please choose a number from the list of names? 1
10  Steve Blenheim
    Number for which name? 4
    Norma Cord
    Number for which name? 5
    Jon DeLoach
    Number for which name? 2
    Betty Boop
    Number for which name? 6
    Karen Evich
    Number for which name? 8
    Number for which name? 3
    Igor Chevsky
    Number for which name?
    <Ctrl>-d or <Ctrl>-z (Exit the program)
```

解释：

1. 打开文件 emp.names，并通过文件句柄 NAMES 访问它。
2. 通过一个 while 循环逐行读取文件内容。
3. 以空白字符（空格、制表符和换行符）作为定界符，将读入的 $_ 内容拆分为两个字段。之后，由 split 函数返回拆分得到的列表内容，即 $num 和 $name（由姓和名构成）。
4. 以 $num 为键，以 $name 作为与该键相关联的值，创建散列变量 %realid。在每次循环时，都向 %realid 添加新的键 / 值对。
5. 关闭文件句柄。
6. 再次进入 while 循环。
7. 要求用户输入与姓名相关联的编号。
8. 从标准输入（此处即为终端键盘）中读取用户键入的数字，并将其赋值给 $num，同时使用 chomp 归整它。
9. 如果 $num 是空值，则退出循环。
10. 打印 %realid 散列变量中与“number”键（即 $num）相关联的“name”值（即 $name）。

5.6.2　特殊散列变量

%ENV 散列变量。%ENV 散列变量中含有父进程（如 shell 或 Web 服务器进程）传给 Perl 的环境变量值。这个散列变量的键是环境变量名，值则是相应的环境变量值。如果改变 %ENV 值的

话，就会改变 Perl 脚本及其所有子进程的环境变量，但不会影响 Perl 的父进程。在 CGI Perl 脚本中，环境变量发挥着重要的作用。CGI 脚本相关内容将在第 16 章介绍。

示例 5.64

```
(In Script)
    #!/usr/bin/perl
1   foreach $key (keys(%ENV){
2      print "$key\n";
    }
3   print "\nYour login name $ENV{'LOGNAME'}\n";
4   $pwd=$ENV{'PWD'};
5   print "\n", $pwd, "\n";
(Output)
2   OPENWINHOME
    MANPATH
    FONTPATH
    LOGNAME
    USER
    TERMCAP
    TERM
    SHELL
    PWD
    HOME
    PATH
    WINDOW_PARENT
    WMGR_ENV_PLACEHOLDER

3   Your login name is ellie

5   /home/jody/home
```

解释：

1. 由 foreach 循环负责遍历% ENV 的所有键。
2. 打印所有键的值。
3. 打印键 LONGNAME 的值。
4. 将键 PWD 的值赋予 $pwd。
5. 打印 $pwd 的值。

%SIG 散列变量。%SIG 散列变量负责为信号指定信号处理程序。譬如在程序运行时，如果用户按下了 <Ctrl>-C，就会产生一个信号，其名称标识为 SIGINT（如需了解完整的信号列表，请读者参考 UNIX 用户手册）。SIGINT 的默认操作是中断当前进程。所谓信号处理程序，是在信号传至进程时自动调用的子例程，一般用于在退出脚本之前执行清除操作或检查某些标记（Flag）值（这里假定所有的信号处理程序都是在 main 包中设置的）。

%SIG 数组中只含有在 Perl 中设置的信号值。

示例 5.65

```
(In Script)
    #!/usr/bin/perl
1   sub handler{
```

```
2        local($sig) = @_;  # First argument is signal name
         print "Caught SIG$sig -- shutting down\n";
         exit(0);
     }
4    $SIG{'INT'} = 'handler';  # Catch <Ctrl>-c
     print "Here I am!\n";
5    sleep(10);
6    $SIG{'INT'}='DEFAULT';
7    $SIG{'INT'}='IGNORE';
       < Program continues here >
```

解释：

1. handler 是子例程的名字。本行定义了一个子例程。
2. $sig 是一个局部变量，含有信号名。
3. 当收到 SIGINT 信号时，显示本消息，然后退出脚本。
4. 键 INT 含有的值是子例程名 handler。因此在信号到达时会调用这个 handler 信号处理程序。
5. sleep 函数给用户 10 秒钟时间，以便用户按下 <Ctrl>-c 并观察发生的现象。
6. 恢复默认的操作。默认操作内容是在用户按下 <Ctrl>-c 时退出执行进程。
7. 如果把“IGNORE”值赋予 $SIG 散列变量的话，则程序会忽略 <Ctrl>-c 并继续执行。

%INC 散列变量。%INC 散列变量负责保存通过 do 或 require 函数导入的文件名路径。其键（key）是文件名；值（value）则是文件的实际位置。

5.6.3 上下文 (Context)

总而言之，在判断变量类型（即“funny”字符）时，Perl 需要依据该变量的使用方式来决定。也就是说，需要根据标量型或列表型的上下文（Context）来决定。

如果出现在赋值运算符左侧的是标量的话，右侧表达式会在标量型上下文中求值；如果左侧变量是数组型的话，右侧的表达式则会在列表型上下文中求值。

有关上下文处理的知识请参阅“从 STDIN 读取”一节。

在本书后面章节中，读者还将看到很多用到上下文的示例。

示例 5.66

```
(The perldoc function describes how reverse works)
1  $ perldoc -f reverse
   reverse LIST
           In list context, returns a list value consisting of the
elements of LIST in the opposite order. In scalar context,
concatenates the elements of LIST and returns a string value with all
characters in the opposite order.
   ......
  (The Perl Script)
   @list = (90,89,78,100,87);
   $str="Hello, world";
   print "Original array: @list\n";4
   print "Original string: $str\n";5
2  @revlist = reverse(@list);
3  $revstr = reverse($str);
4  print "Reversed array is: @revlist\n";
```

```
5   print "Reversed string is: $revstr\n";

6   $newstring = reverse(@list); print "List reversed, context
string: $newstring\n";
(Output)
Original array: 90 89 78 100 87
Original string: Hello, world
Reversed array is: 87 100 78 89 90
Reversed string is: dlrow ,olleH
List reversed, context string: 78001879809
```

解释：

1. 在 Perl 内建的 reverse 函数说明文档中，详细说明了上下文的概念。
2. reverse 函数负责翻转一个数组的所有元素，并将翻转后的元素赋予另一个数组。该操作的上下文是列表型的。
3. 在这里，reverse 函数又负责翻转一个字符串的所有字符，并返回一个标量型的字符串。其上下文是标量型的。
4. 打印翻转后的数组元素。
5. 打印翻转后的字符串内容。
6. 在这里，reverse 又一次对数组进行了翻转，但这次生成的数组则会赋值给一个字符串。由于其上下文是标量型的，因此该函数会翻转数组的所有元素，然后把列表转换为一个字符串。

5.7　读者应当学到的知识

1. 有关数据类型的知识。
2. 如果不给变量赋值的话，Perl 会赋予它什么初始值？
3. 什么是特殊标识（funny symbols）？
4. 双引号中能够解释哪些数据类型？
5. 在一个标量型变量中能保存多少个数字或字符串？
6. “关联数组”又称为什么？
7. 在散列中，可否提供名称相同的键？可否为同一个键关联多个值？
8. 什么是列表（list）？
9. 如何获知数组的长度？
10. 为什么数组元素或散列元素的前缀是 $ ？
11. chop 和 chomp 函数有何区别？
12. 如何只从键盘输入 25 字节内容到一个变量中？
13. splice 和 slice 有哪些区别？
14. 如何对数值型数组进行排序？如何根据值对散列进行排序？
15. 什么函数能同时解析散列的键和值？
16. 能否让两个键拥有相同的名字？
17. %SIG 散列的用途是什么？
18. 什么是环境变量？
19. term 作用域是什么意思？

5.8 下章简介

下一章将探讨 Perl 运算符。我们将介绍不同种类的赋值运算符、比较和逻辑运算符、代数运算符以及位运算符；然后介绍 Perl 是如何处理字符串与数字、如何创建给定范围的数字、以及如何生成随机数的；最后介绍一些特殊的字符串函数。

练习 5 特殊字符

1. 编写一个名为 foods.plx 的脚本，要求用户输入最爱吃的食物，至少输入 5 种。然后把食物列表保存到标量中（这样就无法检查输入了多少种食物了，不过不必在意这个）。
   a) 拆分标量，并创建数组。
   b) 打印数组内容。
   c) 打印数组的第一个和最后一个元素。
   d) 打印数组元素数目。
   e) 根据数组中的三种食物，创建一个数组分片，并打印其内容。
2. 给定一个数组 @names=qw(Nick Susan Chet Dolly Bill)，编写一个 Perl 语句，分别完成下列功能：
   a) 用 Ellie、Beatrice 和 Charles 替换 Susan 与 Chet。
   b) 从数组中删除 Bill。
   c) 把 Lewis 和 Lzzy 添加到数组尾部。
   d) 从数组首部移除 Nick。
   e) 翻转整个数组。
   f) 把 Archie 添加到数组开头。
   g) 排列整个数组。
   h) 移除 Chet 和 Dolly，并用 Christian 和 Daniel 代替它们。
3. 编写一个名为 elective 的脚本，创建如下散列：
   a) 它的键是一些数字编码：2CPR2B、1UNX1B、3SH414、4PL400。
   b) 它的值是一些课程名：C Language、Intro to UNIX、Shell Programming、Perl Programming。
   c) 根据值的顺序对这个散列进行排序，并打印排序结果。
   d) 请用户输入本学期计划选修的课程编号，然后依照下面的格式打印课程内容。
      You will be taking Shell Programming this semester.
4. 修改上述 elective 脚本，使得其输出类似于如下格式。新程序将要求用户输入注册信息，并从菜单中选择一个 EDP 编号。接着程序便会打印相应的课程名。用户在输入 EDP 编号时无需考虑大小写问题。最后，程序会显示一行消息，确认用户的地址并感谢登录。

```
REGISTRATION INFORMATION FOR SPRING QUARTER
Today's date is Wed Apr 19 17:40:19 PDT 2007

Please enter the following information:
Your full name: Fred Z. Stachelin
What is your Social Security Number (xxx–xx–xxxx): 004–34–1234
Your address:
  Street: 1424 Hobart St.
  City, State, Zip:  Chico, CA 95926
```

"EDP" NUMBERS AND ELECTIVES:

____
2CPR2B | C Programming
____
1UNX1B | Intro to UNIX
____
4PL400 | Perl Programming
____
3SH414 | Shell Programming
____

What is the EDP number of the course you wish to take? 4pl400
The course you will be taking is "Perl Programming."

Registration confirmation will be sent to your address at
  1424 HOBART ST.
  CHICO, CA 95926

Thank you, Fred, for enrolling.

5. 编写一个名叫 findem 的脚本，完成如下功能：
   a) 读取文件 datebook 的内容，并赋予一个数组（该文件可在华章网站（www.hzbook.com）上找到）。
   b) 请用户输入待查的姓名。然后使用内建的 grep 函数查询数组中的所有元素，找到含有该姓名的元素，并返回符合条件的元素总数。上述查找过程将忽略大小写。
   c) 使用 split 函数获取当前电话号码。
   d) 使用 splice 函数将当前电话号码替换成新的电话号码，或者利用其他任何内建函数生成如下输出：

Who are you searching for? Karen
What is the new phone number for Karen? 530-222-1255
Karen's phone number is currently 284-758-2857.

Here is the line showing the new phone number:
Karen Evich:530-222-1255:23 Edgecliff Place, Lincoln, NB 92086:7/25/53:85100

Karen was found in the array three times.

6. 编写一个名为 tellme 的脚本，打印 datebook 文件中的所有姓名、电话和薪酬数据。若要执行该脚本，请键入如下命令：
   tellme datebook
   其输入应当遵循如下形式：
   Salary: 14500
   Name: Betty Boop
   Phone: 245–836–8357

第 6 章　运　算　符

6.1　关于 Perl 运算符

在现实世界中，操作员（operator）负责操作配电盘、计算机、推土机、坦克等机械。而在 Perl 中，运算符（operator）则负责操纵数字、字符串或者二者的组合。运算符是一些能根据特定的规则产生计算结果的符号，譬如 +、－、=、>、< 等。运算符处理的数据对象即为操作数（oprand），譬如在表达式 5 ＋ 4 中，5 和 4 都是操作数。表达式中含有运算符和操作数。如果在表达式末尾加上分号，就得到了一个独立的语句；譬如：n=5+4。

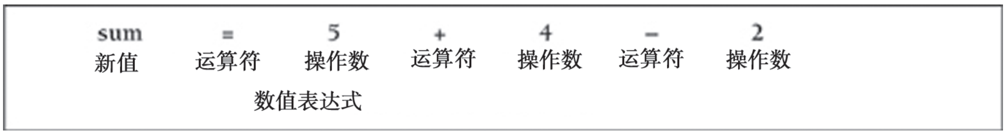


图 6-1　计算一个表达式

数值表达式 5 ＋ 4 － 2 中含有三个操作数，其运算符是＋号和－号。其中，＋号对应的操作数是 5 和 4，该运算完成后，表达式就变成了 9 － 2。全部计算完毕后，整个表达式的值是 7。由于加号和减号运算符都只能处理两个操作数，因此它们又称作是二元运算符（binary operator）。如果能处理单个操作数的话，就称为一元运算符（unary operator）；如果能处理三个的话，则称为三元运算符（ternary operator）。本章将给出这些不同类型运算符的示例。

大部分的 Perl 运算符都继承自 C 语言，不过 Perl 也提供了自己独特的运算符。[⊖]

6.2　混合数据类型

如果操作数属于混合类型（譬如数字与字符串），Perl 会首先判断运算符期望的操作数类型，然后作相应的类型转换。这个过程又称为运算符重载（overload）。

如果运算符是数值型的（如代数运算符），而操作数却是字符串的话，Perl 将把该字符串转换

⊖ 运算符既可以是符号，也可以是字符。在 Perl 5 中，如果省略了函数后面的括号，则脚本会把它当作运算符来使用。

为相应的十进制浮点数。未定义的字符串值会转换为数字 0。如果字符串首部拥有空格前缀，或者末尾带有非数字字符的话，Perl 会忽略它们。如果某个字符串无法转换为相应数字的话，Perl 会将其变成 0。

```
$string1 = "5 dogs ";
$string2 = 4;
$number = $string1 + $string2;  # Numeric context
print "Number is $number.\n";   # Result is  9
```

同样，如果 Perl 碰到了字符串型运算符，而相应操作数是数值型的话，便会将数字转换为字符串进行处理。例如，下面使用了连接运算符，负责把两个字符串连接到一起。

```
$number1 = 55;
$number2 = "22";
$string = $number1 . $number2;  # Context is string
print "String is string.\n"     # Result is "5522"
```

表 6-1　如何把字符串转换为数字

字符串	——→ 转换为 ——→	数字
"123 go!"		123
"hi therev"		0
"4e3"		4000
"-6**3xyz"		-6
".456!!"		0.456
"x.1234"		0
"0xf"		0

示例 6.1

```
(The Script)
1   $x = "     12hello!!" + "4abc\n";
# Perl will remove leading whitespace and trailing non-numeric
# characters
2   print "$x";
3   print "\n";

4   $y = ZAP . 5.5;
5   print "$y\n";

(Output)
2   16
5   ZAP5.5
```

解释：

1. 加号（＋）是一个运算符。字符串“12hello”和“4abc\n”会转换成数字（需去除其空格前缀和末尾的非数字字符），然后再执行加法运算。其运算结果存储到标量 $x 中。
2. 打印标量 $x。
3. 由于在转换成数字时会把 \n 从字符串 4\n 中去除，因此在打印时需要提供另一个换行符 \n。

4. 点号（.）左右都是空白字符，因此它是一个字符串运算符，作用是连接两个字符串。在本行中，数字 5.5 将先转换成字符串，然后和另一个字符串 ZAP 连接在一起。
5. 打印标量 $y 的值。

6.3 优先级和结合性

当表达式中含有多个运算符和操作数时，运算结果就有不止一种可能。这时就需要通过优先级和结合性来告诉编译器如何对这种表达式求值。所谓优先级（Precedence），指的是运算符结合操作数的能力。譬如，乘法运算符结合操作数的能力要强于加法运算符，因此具有更高的优先级。赋值运算符则拥有较低的优先级，因此它与操作数的结合较为松散[⊖]。括号拥有最高的优先级，因而可以用于控制表达式的求值顺序。当多个括号嵌套出现时，位于内层的括号拥有更高的优先级。

所谓结合性（Asscociativity），指的是运算符计算操作数的顺序：从左到右、未规定顺序、或者从右到左。

在下面这个示例中，表达式是如何求值的呢？是先进行加法、乘法？还是先进行除法？又是以何种顺序呢：从左到右还是从右到左？

示例 6.2

```
(The Script)
1   $x = 5 + 4  * 12 / 4;
2   print "The result is  $x\n";

(Output)
2   The result is 17
```

解释：

1. 本行的结合性是从左到右。乘法和除法的优先级高于加法与减法，而加法、减法的优先级又高于赋值运算符。为了说明上述关系，下面用括号组合了各个操作数。事实上，如果需要强制指定优先级，用户应当在操作数两边加上括号，并按照期望的求值顺序对操作数进行组合。
   ```
   $x = ( 5 + ( ( 4 * 12 ) / 4 ) );
   ```
2. 对表达式求值，并打印结果到 STDOUT。

表 6-2 总结了 Perl 运算符的优先级和结合性规则。表中位于同一行的运算符具有相同的优先级，越靠前的行优先级越高。

表 6-2 优先级和结合性

运算符	描述信息	结合性
() [] {}	函数调用，数组下标	左到右
->	反引用运算符	左到右

⊖ 记住优先级规则的一个简单方法：Please Excuse My Dear Aunt Sally。它分别表示括号、乘幂、乘法、除法、加法和减法运算符。

（续）

运算符	描述信息	结合性
++ --	自动递增、递减	无
**	乘幂	右到左
! ~ \ + -	逻辑 not、位 not、一元加法、一元减法	右到左
=~ !~	匹配、不匹配	左到右
* / % x	乘、除、模、字符串重复	左到右
+ - .	加、减、字符串连接	左到右
<< >>	按位左移、右移	左到右
-r -w -x -o 等运算符	有名字的一元运算符，如文件测试运算符等	无
< <= > >= lt le gt ge	数字和字符串测试，如小于、大于	无
== != <=> eq ne cmp	数字和字符串测试，如等于、不等于	无
&	按位 AND	左到右
| ^	按位 OR 或 XOR	左到右
&&	逻辑与 AND	左到右
||	逻辑或 OR	左到右
..	范围运算符	无
?:	三元条件运算符	右到左
= += -= *= /= %=	赋值运算符	右到左
, =>	求左值，丢弃之，并求右值	左到右
not	等效于！，但具有较低优先级	右
and	等效于 &&	左到右
or xor	等效于 || 和 ^	左到右

6.3.1 赋值运算符

等于号（＝）是一种赋值运算符，能将等于号右侧的值赋予左边的变量。表 6-3 展示了继承自 C 语言的赋值和快速赋值语句。

表 6-3 赋值运算符

运算符	示例	含 义
=	$var = 5;	把 5 赋予变量 $val
+=	$var += 3;	把 $val 值加上 3，并把结果再存入 $val
-=	$var -= 2;	把 $val 值减去 2，并把结果再存入 $val

（续）

运算符	示例	含　义
.=	$str .= "ing";	将原字符串与“ing”连接，并将结果赋予 $str
*=	$var *= 4;	把 $val 值乘以 4，并把结果再存入 $val
/=	$var /= 2;	把 $val 值除以 2，并把结果再存入 $val
**=	$var **= 2;	把 $val 值平方后再存入 $val
%=	$var %= 2;	把 $val 值除以 2，并把余数存入 $val
x=	$str x= 20;	把字符串 $str 重复 20 次后存入 $str
<<=	$var <<= 1;	把 $val 左移 1 位，并把结果再存入 $val
>>=	$var >>= 2;	把 $val 右移 2 位，并把结果再存入 $val
&=	$var &= 1;	把 $val 值和 1 逐位 AND，并把结果再存入 $val
|=	$var |= 2;	把 $val 值和 2 逐位 OR，并把结果再存入 $val
^=	$var ^= 2;	把 $val 值和 2 逐位 XOR，并把结果再存入 $val

示例 6.3

```
(The Script)
    #!/usr/bin/perl
1   $name="Dan";
    $line="*";
    $var=0;        # Assign 0 to var

2   $var += 3;     # Add 3 to $var; same as $var=$var+3
    print  "\$var += 3 is $var \n";

3   $var -= 1;     # Subtract 1 from $var
    print "\$var -= 1 is $var\n";

4   $var **= 2;         # Square $var
    print "\$var squared is $var\n";

5   $var %= 3;          # Modulus
    print "The remainder of \$var/3 is $var\n";

6   $name .= "ielle";  # Concatenate string "Dan" and "ielle"
    print "$name is the girl's version of Dan.\n";
7   $line x= 10;        # Repetition; print 10 stars
    print "$line\n";

8   printf "\$var is %.2f\n", $var=4.2 + 4.69;

(Output)
2   $var += 3 is 3
3   $var -=1 is 2
4   $var squared is 4
5   The remainder of $var/3 is 1
6   Danielle is the girl's version of Dan.
```

```
7   **********
8   $var is 8.89
```

解释：

1. 将等于号右侧值赋值给等于号左侧的标量型变量。
2. 赋值运算符 += 负责给标量 $var 加 3，等效于 $var = $var+3。
3. 赋值运算符 -= 负责给标量 $var 减 1，等效于 $var = $var-1。
4. 赋值运算符 **= 负责给标量 $var 计算平方值，等效于 $var = $var**2。
5. 运算符 % 负责计算标量 $var 除以 3 的余数。该运算符又称作模数运算符或者余数运算符。表达式 $var%=3 等效于 $var=$var%3。
6. 运算符 . 负责将字符串“ielle”与标量 $name 相连，等效于 $name=$name."ielle"。
7. 重复运算符是一个二元运算符。其右侧的操作数是左边字符串型操作数的重复次数。标量 $line 的值是将星号（*）重复 10 次。
8. 使用 printf 函数格式化并打印两个浮点数的和。

6.3.2　关系运算符

关系运算符用于比较操作数的逻辑关系，其比较结果是逻辑真（true）或者假（false）⊖。Perl 提供了两类关系运算符，其中一类用于比较数字，另一类则负责比较字符串。用户在碰到 if/else、while 循环等语法结构时往往需要使用这些关系运算符。譬如：

```
if ( $x > $b ) { print " $x is greater.\n" ; }
```

本书将在第 7 章讨论条件量相关知识。

表达式（5>4>2）会抛出语法错误，因为它不符合结合性要求（详见表 6-2）。

数值型关系运算符。表 6-4 列举了所有数值型的关系运算符。

表 6-4　关系运算符与数值

运算符	示例	含义
>	$x > $y	$x 大于 $y
>=	$x >= $y	$x 大于或等于 $y
<	$x < $y	$x 小于 $y
<=	$x <= $y	$x 小于或等于 $y

示例 6.4

```
(The Script)
    $x = 5;
    $y = 4;
    $result = $x > $y;
    print "$result\n";
```

⊖ 和其他多种编程语言一样，Perl 也不支持特别的布尔型数据类型，而是通过数值 1 或 0 来表示“true”和“false”。不过，任何非 0 的表达式值都可以表示 true，而！ 1 则表示 false；譬如：while(1) 或者 if !1 等。

```
3   $result = $x < $y;
4   print $result;

(Output)
2   1
4   0
```

解释：

1. 如果 $x 大于 $y，则返回 1（真），并将其保存到 $result 中，否则返回 0（假）。
2. 表达式的值是真，因此把 $result 的值 1 打印到 STDOUT。
3. 如果 $x 小于 $y，则返回 1（真），并将其保存到 $result 中，否则返回 0（假）。
4. 表达式的值是假，因此把 $result 的值 0 打印到 STDOUT。

字符串型关系运算符。字符串型关系运算符通过把前一个字符串操作数中每个字符的 ASCII 码值与后一个字符串中对应字符的码值相比较，从而完成对两个操作数的比较。上述比较过程将包括字符串末尾的空白字符。

如果前一个字符串中某个字符的 ASCII 码值比后一字符串中相应字符的码值高的话，返回 1；否则就返回 0。

表 6-5 列举了所有字符串型关系运算符。

表 6-5　关系运算符与字符串值

运算符	示例	含义
gt	$str1 gt $str2	$str1 大于 $str2
ge	$str1 ge $str2	$str1 大于或等于 $str2
lt	$str1 lt $str2	$str1 小于 $str2
le	$str1 le $str2	$str1 小于或等于 $str2

示例 6.5

```
(The Script)
1   $fruit1 = "pear";
2   $fruit2 = "peaR";
3   $result = $fruit1 gt $fruit2;
4   print "$result\n";

5   $result = $fruit1 lt $fruit2;
6   print "$result\n";

(Output)
4   1
6   0
```

解释：

1. 把字符串 pear 赋值给标量 $fruit1。
2. 把字符串 perR 赋值给标量 $fruit2。
3. 在比较 $fruit1 和 $fruit2 中的每个字符时，会发现 r 与 R 之前的每个字符都相等。小写字母 r 的 ASCII 码值是 114，而大写字母 R 的码值则是 82。由于 114 大于 82，因此字符串比较结果是 1（真）。

4. 由于表达式为真，因此把 $result 值 1 打印到 STDOUT。
5. 与上述第 3 行情况正好相反。由于大写字母 R 的 ASCII 码值（82）小于小写字母 r 的码值（114），因此字符串比较结果是 0（假），即 peaR 小于 pear。
6. 由于表达式为假，因此把 $result 值 0 打印到 STDOUT。

6.3.3 相等性运算符

相等性运算符既能处理数值型操作数，也能处理字符串型操作数（详见表 6-6 和表 6-7）。但请读者务必在比较数字时使用数值型相等性运算符，在比较字符串时则应使用字符串型相等性运算符。例如，如果读者给出这个表达式：

```
"5 cats " == "5 dogs"
```

则该表达式的值将是 true。为什么？这是因为 Perl 看到的是一个数值型相等性运算符 ==，期望其操作数都是数字，而不是字符串。因此，Perl 会把“5 cats”转换成 5，同时把“5 dogs”也转换成数字 5，显然 5 等于 5。故而上述表达式值为真。（在进行转换时，Perl 会从字符串的左边开始扫描数字。如果碰到的字符是数字，就保存它；如果碰到了非数字字符，则退出转换过程。）

表 6-6 相等性运算符与数字值

运算符	示例	含 义
==	$num1 == $num2	$num1 等于 $num2
!=	$num1 != $num2	$num1 不等于 $num2
<=>	$num1 <=> $num2	比较 $num1 和 $num2 的大小。
		如果 $num1 大于 $num2，返回 1；
		如果 $num1 等于 $num2，返回 0；
		如果 $num1 小于 $num2，则返回 -1

表 6-7 相等性运算符与字符串值

运算符	示例	含 义
eq	$str1 eq $str2	$str1 等于 $str2
ne	$str1 ne $str2	$str1 不等于 $str2
cmp	$str1 cmp $str2	比较 $str1 和 $str2 并返回带符号整数

数值型相等性运算符。数值型相等性运算符通过比较操作数（数字）的大小来返回表达式值。如果数字相等的话，返回真（1）；如果不相等，则返回假（0）。

数值型比较运算符则比较两个操作数的大小，如果第一个操作数小于第二个操作数，就返回－1；如果相等则返回 0；如果第一个操作数大于第二个操作数，就返回 1。

示例 6.6

```
(The Script)
    $x = 5;
```

```
     $y = 4;
1    $result = $x == $y;
2    print "$result\n";

3    $result = $x != $y;
4    print "$result\n";
5    $result = $x <=> $y;
6    print "$result\n";

7    $result = $y <=> $x;
8    print "$result\n";

(Output)
2    0
4    1
6    1
8    -1
```

解释：

1. 如果 $x 等于 $y，就返回 1（真），并将其保存到 $result 中，否则返回 0（假）。
2. 由于表达式的值是假，因此把 $result 的值 0 打印到 STDOUT。
3. 如果 $x 不等于 $y，就返回 1（真），并将其保存到 $result 中，否则返回 0（假）。
4. 由于表达式的值是真，因此把 $result 的值 1 打印到 STDOUT。
5. 比较标量 $x 和 $y 的值。如果 $x 大于 $y，返回 1；如果 $x 等于 $y，返回 0；如果 $x 小于 $y，则返回有符号整数 -1。
6. 由于 $x 大于 $y，因此把 $result 的值 1 打印到 STDOUT。
7. 比较标量 $y 和 $x 的值。如果 $y 大于 $x，返回 1；如果 $y 等于 $x，返回 0；如果 $y 小于 $x，则返回有符号整数 -1。
8. 由于 $x 小于 $y，因此把 $result 的值 -1 打印到 STDOUT。

字符串型相等性运算符。字符串型相等性运算符通过把第一个操作数中每个字符的 ASCII 码值与第二个操作数相应字符的码值相比较，从而完成对两个操作数的比较。上述比较过程将包括字符串末尾的空白字符。

如果前一个字符串中某个字符的 ASCII 码值比后一字符串中相应字符的码值高的话，返回 1；如果两个字符串的所有字符码值相等的话，就返回 0；如果前一个字符串中某个字符的 ASCII 码值比后一字符串中相应字符的码值低的话，则返回 -1。

示例 6.7

```
(The Script)
1    $str1 = "A";
     $str2 = "C";
     $result = $str1 eq $str2;
     print "$result\n";

2    $result = $str1 ne $str2;
     print "$result\n";

3    $result = $str1 cmp $str2;
     print "$result\n";
```

```
4   $result = $str2 cmp $str1;
    print "$result\n";

5   $str1 = "C";        # Now both strings are equal
6   $result = $str1 cmp $str2;
    print "$result\n";

(Output)
1   0
2   1
3   -1
4   1
6   0
```

解释：

1. 把标量 $str1 赋值为 A，把标量 $str2 赋值为 C。如果 $str1 等于 $str2 的话，返回真（1）。将它赋值给变量 $result，并打印该变量的值。
2. 如果 $str1 不等于 $str2，则返回真（1）。将它赋值给 $result，并打印该变量值。
3. 当比较 $str1 和 $str2（即对字符串中的每个字符进行比较）时，如果所有的字符都相等，把返回值 0 赋予变量 $result。如果 $str1 大于 $str2，返回 1；如果 $str1 小于 $str2，返回 -1。本例中，$str1 是小于 $str2 的。然后打印 $result 的值。
4. 这里翻转了比较的顺序。由于 $str2 是大于 $str1 的，因此结果是 1。然后打印 $result 的值。
5. 将 $str1 赋值为 C，即和 $str2 的值相同。
6. 现在 $str1 是等于 $str2 的。由于它们的所有字符都相同，因此将返回 0，并赋给 $result。最后打印 $result 的值。

示例 6.8

```
(The Script)
    # Don't use == when you should use eq!
1   $x = "yes";
    $y = "no";
    print "\nIs yes equal to no? If so, say 1; if not say 'null'.\n";
2   print "The result is: ",$x == $y,"\n";     # Should be $x eq $y

(Output)
1   Is yes equal to no? If so, say 1; if not say 'null'.
2   The result is: 1
```

解释：

1. 分别将标量 $x 和 $y 赋值为字符串 yes 和 no。
2. 这里错误地使用数值型相等性运算符 == 处理两个字符串。Perl 将把这两个字符串转换成数字。由于其中不包含任何数字字符，因此这两个字符串都会转换为 0（零）。由于 0 等于 0，因此得到结果是 1（真）。事实上应当在这里使用字符串型相等性运算符 eq。

6.3.4　逻辑运算符（短路运算符）

短路运算符会从左到右依次测试每个操作数的真假与否。当满足一定的真假条件后，就不再进行进一步求值。与 C 语言一样，Perl 的短路运算符不返回 0（假）或者 1（真），而是返回最后一个

操作数的值。用户一般会在条件语句（详见第 7 章“条件语句”）中用到这些运算符。

如果运算符 && 左侧的表达式值为 0，则整个表达式的值就是 0。如果该运算符左侧表达式的值是真（非 0）的话，才继续对右侧表达式求值并返回其值。

逻辑运算符也可通过 and、or 或 not 来表达，但是后者的优先级较低。详情请参阅表 6-2。如果运算符 || 左侧的表达式值为真（非 0），即立刻返回该表达式的值。如果其左侧表达式的值为假，才会继续求出运算符右侧表达式的值并返回之。

表 6-8 列出了所有逻辑运算符。

表 6-8　逻辑运算符（短路运算符）

运算符	替代格式	示例	含　义
&&	and	$x && $y	如果 $x 为真，则评价并返回 $y 的值
		$x and $y	如果 $x 为假，则评价并返回 $x 的值
||	or	$x || $y	如果 $x 为真，则评价并返回 $x 的值
		$x or $y	如果 $x 为假，则评价并返回 $y 的值
	xor	$x xor $y	当 $x 和 $y 中有且仅有一个为真时，返回真
!	not	!$x	非 $x。 当 $x 为假时返回真。
		not $x	

示例 6.9

```
(The Script)
    #!/usr/bin/perl
    # Short-circuit operators
1   $num1=50;
2   $num2=100;
3   $num3=0;

4   print $num1 && $num3, "\n";     # result is 0
5   print $num3 && $num1, "\n";     # result is 0
6   print $num1 && $num2, "\n";     # result is 100
7   print $num2 && $num1, "\n\n";   # result is 50

8   print $num1 || $num3, "\n";     # result is 50
9   print $num3 || $num1, "\n";     # result is 50
10  print $num1 || $num2, "\n";     # result is 50
11  print $num2 || $num1, "\n";     # result is 100

(Output)
4   0
5   0
6   100
7   50
8   50
9   50
10  50
11  100
```

解释：

1. 把标量 $num1 赋值为 50。
2. 把标量 $num2 赋值为 100。
3. 把标量 $num3 赋值为 0。
4. 由于 && 运算符左侧表达式 $num1 的值不是 0（即为真），返回 && 运算符右侧表达式 $num3 的值。
5. 由于 && 运算符左侧表达式 $num3 的值为 0（假），返回表达式 $num3 的值。
6. 由于 && 运算符左侧表达式 $num1 的值为真，返回 && 右侧表达式 $num2 的值。
7. 由于 && 运算符左侧表达式 $num2 的值为真，返回 && 右侧表达式 $num1 的值。
8. 由于 || 运算符左侧表达式 $num1 的值非 0（即为真），返回表达式 $num1 的值。
9. 由于 || 运算符左侧表达式 $num3 的值为 0（即为假），返回 || 右侧表达式 $num1 的值。
10. 由于 || 运算符左侧表达式 $num1 的值非 0（即为真），返回表达式 $num1 的值。
11. 由于 || 运算符左侧表达式 $num2 的值非 0（即为真），返回表达式 $num2 的值。

6.3.5 逻辑字运算符

本节将要介绍的这些逻辑运算符的优先级要低于短路运算符，但其功能基本相同。它们能让程序更加易于理解，并且也具有短路特性。除了常见的短路运算符外，逻辑字运算符还增添了一个 xor（异或）运算符。

示例 6.10

```
    # Examples using the word operators
1   $num1=50;
    $num2=100;
    $num3=0;
    print "\nOutput using the word operators.\n\n";
2   print "\n$num1 and $num2: ",($num1 and $num2);, "\n";
3   print "\n$num1 or $num3: ", ($num1 or $num3), "\n";
4   print "\n$num1 xor $num3: ",($num1 xor $num3), "\n";
5   print "\nnot $num3: ", not $num3;
    print "\n";

(Output)
    Output using the word operators.

2   50 and 100: 100

3   50 or 0: 50

4   50 xor 0: 1

5   not 0: 1
```

解释：

1. 为标量 $num1、$num2 和 $num3 赋初值。
2. and 运算符能评价其两个操作数的值。由于 $num1 和 $num2 都为真（true），因此会返回后一个操作数的值，得到 100。由于 100 不等于 0，因此整个表达式的值为真。
3. or 运算符评价其两个操作数的值。因为 $num1 为真，而逻辑字运算符也有短路性，因此当第一个表达式为真时就不必继续求值。其返回值是 50，也为真。

本 PDF 书签由 龙睿·LoRui 制作。
www.LoRui.com

来看看这些东东……

- ✧ Yahoo!疯了！
- ✧ HP笔记本激活Windows 7

域名转让：

- ✓ DaNanPing.com（大南平）
- ✓ 15500.net
- ✓ Gn7c.com
- ✓ DaLongNan.com（大陇南、大龙南）
- ✓ 51perl.com（无忧 Perl）
- ✓ LoRui.com

4. xor（异或）运算符评价其两个操作数的值，它没有短路性。如果其中有且仅有一个操作数为真，就返回 1；如果两个操作数都是真或都是假，则返回假。
5. 逻辑 not 运算符只处理其右侧的操作数。如果操作数为真，返回假；如果操作数为假，则返回真。

示例 6.11

```
    (The Script)
    # Precedence with word operators and short-circuit operators
    $x=5;
    $y=6;
    $z=0;
1   $result=$x && $y && $z;        # Precedence of = lower than &&
    print "Result: $result\n";

2   $result2 = $x and $y and $z;  # Precedence of = higher than and
    print "Result: $result2\n";

3   $result3 = ( $x and $y and $z );
    print "Result: $result3\n";

(Output)
1   Result: 0
2   Result: 5
3   Result: 0
```

解释：

1. 逻辑短路运算符能评价每个表达式，并返回最后一个表达式的值。本行把变量 $result 赋值为 0。由于 && 运算符比等于号的优先级高，因此先由逻辑运算符对该表达式求值。
2. 这里用到了逻辑字运算符，它的优先级低于等于号。本行将把等于号右侧第一个表达式的值赋予 $result2。
3. 通过为等于号右侧的表达式添加括号，便能先求该表达式的值，并将其结果赋值给 $result3。

6.3.6　算术运算符

表 6-9 列出了 Perl 提供的所有算术运算符。

表 6-9　算术运算符

运算符	示例	含义
+	$x+$y	加法
-	$x-$y	减法
*	$x*$y	乘法
/	$x/$y	除法
%	$x%$y	模运算（求余数）
**	$x**$y	乘幂

示例 6.12

```
(The Script)
1   printf "%d\n", 4 * 5 / 2;
2   printf "%d\n", 5 ** 3;
3   printf "%d\n", 5 + 4 - 2 * 10;
4   printf "%d\n", (5 + 4 - 2 ) * 10;
5   printf "%d\n", 11 % 2;

(Output)
1   10
2   125
3   -11
4   70
5   1
```

解释：

1. printf 函数会把算术表达式的计算结果格式化为十进制数。本行执行了乘法和除法运算。这两个运算符的优先级相同，结合性为从左到右，等效于 (4*5)/2。
2. printf 函数会把算术表达式的计算结果格式化为十进制数。乘幂运算符负责计算操作数 5 的三次方，等效于 5^3。
3. printf 函数会把算术表达式的计算结果格式化为十进制数。由于乘法运算符的优先级高于减法，因此首先进行乘法运算，在从左到右结合。等效于 5+4-(2*10)。
4. printf 函数会把算术表达式的计算结果格式化为十进制数。由于括号具有最高优先级，因此这里首先计算位于括号中的乘幂操作。
5. printf 函数会把算术表达式的计算结果格式化为十进制数。模数运算符负责获得两个操作数相除的余数。

6.3.7　自动递增与自动递减运算符

自动递增和自动递减运算符继承自 C 语言（详见表 6-10）。自动递增运算符能把操作数加 1，而自动递减运算符则能对操作数减 1。当用于单个变量时，这两种运算符只不过是传统的加 1 或减 1 操作的快捷方式而已。但当把它们用于赋值语句，或者与其他运算符结合使用时，其最终运算结果就取决于运算符的具体位置了（详见表 6-11）。

表 6-10　自动递增和自动递减运算符

示例	描述	等价形式
++$x	前递增	$x = $x + 1
$x++	后递增	$x = $x + 1
--$x	前递减	$x = $x − 1
$x--	后递减	$x = $x − 1

表 6-11　自动递增和自动递减运算符与赋值运算

示例	描述	等价形式	结果
如果 $x 和 $y 都为 0：$y=$x++	将 $x 的值赋予 $y，然后递增 $x	$y=$x;$x=$x+1;	$y 为 0；$x 等于 1

（续）

示例	描述	等价形式	结果
如果 $x 和 $y 都为 0：$y=++$x	先递增 $x，然后将 $x 的值赋予 $y	$x=$x+1;$y=$x;	$y 为 1；$x 等于 1
如果 $x 和 $y 都为 0：$y=$x--	将 $x 的值赋予 $y，然后递减 $x	$y=$x;$x=$x-1;	$y 为 0；$x 等于 -1
如果 $x 和 $y 都为 0：$y=--$x	先递减 $x，然后将 $x 的值赋予 $y	$x=$x-1;$y=$x;	$y 为 -1；$x 等于 -1

示例 6.13

```
(The Script)
    #!/usr/bin/perl
1   $x=5; $y=0;
2   $y=++$x;      # Add 1 to $x first; then assign to $y
3   print "Pre-increment:\n";
4   print "y is $y\n";
5   print "x is $x\n";
6   print  "-----------------------\n";
7   $x=5;
8   $y=0;
9   print "Post-increment:\n";
10  $y=$x++;      # Assign value in $x to $y; then add 1 to $x
11  print "y is $y\n";
12  print "x is $x\n";

(Output)
3   Pre-increment:
4   y is 6
5   x is 6
    -----------------------
9   Post-increment
11  y is 5
12  x is 6
```

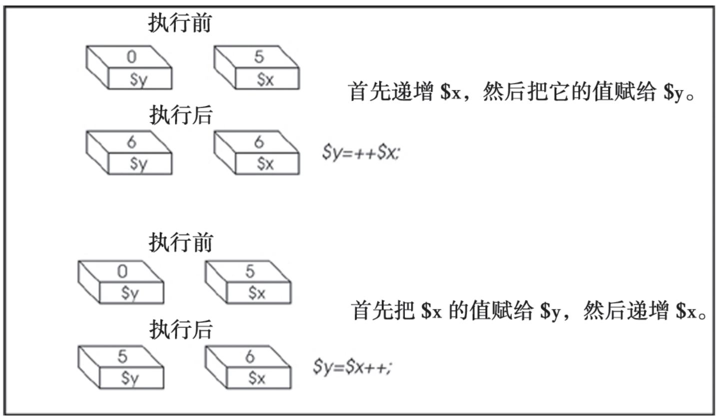


图 6-2　前递增和后递增运算符

6.3.8 位逻辑运算符

位运算简介。人们一般都用十进制来表示数字，十进制数以 10 作为基数，其范围是从 0 到 9，譬如：$100000 以及 1955 等。HTML 颜色编码则是十六进制格式的，其基数为 16，数字范围从 0 到 15，譬如 #00FFFF 代表青色，而 #FF00FF 代表洋红色。计算机则是以二进制形式保存所有信息的，其基数是 2。二进制数值系统仅使用 0 和 1 这两个数字就能表达所有数，其中每个 0 或者 1 都称作是一个“位”（bit）。读者看到的所有信息都是以位的形式存储的。一个字节由 8 个位组成，一个字（word）由 2 个字节或 16 个位组成。最后，2 个字在一起可以组成一个双字（dword），它由 32 个位组成。计算机只用 0 和 1 来表达所有信息，这是因为可以用电流出现与否来表达这两个值。倘若芯片上的电流达到了某个值，就表示 1，否则就是 0。只用两个数字的话，构建计算机硬件会比用更多数字（如 10 个或者 16 个）显得更容易也更便宜。因此计算机以二进制方式保存所有信息。

位运算符。目前几乎所有的处理器都是基于 32 位数构建的。例如，Win32 这个名称就来源于其 Win32 编译器的默认字长是 32 位。位运算符允许用户打开或关闭某个整数内的特定位。例如，如果要为文件设定一个只读标志位，用户只需设定两个值：开启（on）或关闭（off），表示为 1 或 0。如果左右参数均为字符串的话，位操作符便会对字符串内各个字符进行处理。

位运算符将其操作数视为一组 32 个位（0 和 1），而非十进制、十六进制或八进制数字。例如，十进制数 9 的二进制形式是 1001。尽管位运算符是以位的形式执行运算的，但其最终结果则是标准的 Perl 数值形式（参见示例 6.13）。如果读者需要处理图像、游戏、加密、注册表、或设置开关等其他需要进行“纯位运算”的操作，就会感受到位运算符的有用之处了。一般而言，此类运算应当用更高级的编程语言（如 C 和 Java）来处理。

如果读者准备在位层面处理整数值的话，就需要用到位逻辑运算符。位逻辑运算符是一系列二元运算符，负责以内部二进制格式处理位于其两侧的操作数。位运算符会对操作数逐位进行比较，并产生相应的二进制值（详见表 6-12 和表 6-13）。

表 6-12 位逻辑运算符

运算符	示例	含义
&	$x & $y	逐位与
|	$x | $y	逐位或
^	$x ^ $y	逐位异或
<<	$x << 1	逐位左移，整数乘以 2
>>	$x >> 1	逐位右移，整数除以 2

表 6-13 位运算符的执行结果

$x	$y	$x & $y	$x | $y	$x ^ $y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

示例 6.14

```
(The Script)
1   print 5 & 4,"\n";        # 101 & 100
2   print 5 & 0,"\n";        # 101 & 000
3   print 4 & 0,"\n";        # 100 & 000
4   print 0 & 4,"\n";        # 000 & 100
5   print "=" x 10,"\n";     # print 10 equal signs
6   print 1 | 4,"\n";        # 001 & 100
7   print 5 | 0,"\n";        # 101 | 000
8   print 4 | 0,"\n";        # 100 | 000
9   print 0 | 4,"\n";        # 000 | 100
    print "=" x 10,"\n";     # print 10 equal signs
10  print 5 ^ 4,"\n";        # 101 ^ 100
11  print 5 ^ 0,"\n";        # 101 ^ 000
12  print 4 ^ 0,"\n";        # 100 ^ 000
13  print 0 ^ 4,"\n";        # 000 ^ 100

(Output)
1   4
2   0
3   0
4   0
5   ==========
6   5
7   5
8   4
9   4
    ==========
10  1
11  5
12  4
13  4
```

解释：

1. 5 逐位与 4 得到二进制值 100，即十进制值 4。
2. 5 逐位与 0 得到二进制值 000，即十进制值 0。
3. 4 逐位与 0 得到二进制值 000，即十进制值 0。
4. 0 逐位与 4 得到二进制值 000，即十进制值 0。
5. x 运算符让 print 函数打印 10 个等于号。
6. 1 逐位或 4 得到二进制值 101，即十进制值 5。
7. 5 逐位或 0 得到二进制值 101，即十进制值 5。
8. 4 逐位或 0 得到二进制值 100，即十进制值 4。
9. 0 逐位或 4 得到二进制值 100，即十进制值 4。
10. 5 逐位异或 4 得到二进制值 001，即十进制值 1。
11. 5 逐位异或 0 得到二进制值 101，即十进制值 5。
12. 4 逐位异或 0 得到二进制值 100，即十进制值 4。
13. 0 逐位异或 4 得到二进制值 100，即十进制值 4。

示例 6.15

```
(The Script)
```

```
    #!/usr/bin/perl
    # Convert a number to binary
1   while (1) {
2       $mask = 0x80000000;      # 32-bit machine
3       printf("Enter an unsigned integer: ");
4       chomp($num=<STDIN>);
5       printf("Binary of %x hex is: ", $num);
6       for ($j = 0; $j < 32; $j++) {
7          $bit = ($mask & $num) ? 1 : 0;
8          printf("%d", $bit);
9          if ($j == 15)){
10             printf("--");
           }
11     $mask /=2;              # $mask >>= 1; not portable
       }
       printf("\n");
    }
(Output)
Enter an unsigned integer: 1
Binary of 1 hex is: 0000000000000000--0000000000000001
Enter an unsigned integer: 5
Binary of 5 hex is: 0000000000000000--0000000000000101
Enter an unsigned integer: 10
Binary of a hex is: 0000000000000000--0000000000001010
Enter an unsigned integer: 12
Binary of c hex is: 0000000000000000--0000000000001100
Enter an unsigned integer: 15
Binary of f hex is: 0000000000000000--0000000000001111
Enter an unsigned integer: 200
Binary of c8 hex is: 0000000000000000--0000000011001000
```

解释：

1. 这个小程序引入了一些尚未介绍的结构，但这里本意是为了展示如何在完成实际任务时使用上述逐位操作（在此将把数字转换为二进制数并打印）。从第一行开始循环执行，直到用户按下 <Ctrl>-c（在 UNIX 下）或 <Ctrl>-d（在 Windows 下）为止。
2. 把标量值设置为表示 32 个 0 的十六进制值。本示例只能运行在 32 位计算机上。
3. 要求用户输入一个整数。
4. 把该数值赋予变量，并执行 chomp 换行符。
5. printf 函数将以十六进制格式打印变量值。
6. for 循环重复迭代 32 次，每次处理一位。
7. 用 $mask 对 $num 逐位进行与运算。如果结果是 1，则把 1 赋值给 $bit；否则就把 0 赋值给它（详见后面的“条件运算符”一节内容）。
8. 打印 $bit 的值。
9. 如果 $j 的值是 15 的话（表明该循环已经迭代了 16 次），打印两个下划线。
10. 用 $mask 的值除以 2。该操作等效于右移 1 位，但不会移动符号位。

6.3.9　条件运算符

条件运算符是另一种继承自 C 语言的语法结构。它带有三个操作数，因此又称作三元运算符。它是 if/else 结构的简便形式。

格式：

```
conditional expression ? expression : expression
```

示例 6.16

```
$x ? $y : $z
```

解释：

如果 $x 值为真（true）的话，表达式取值为 $y 的值。如果 $x 值为假（false）的话，表达式则取值为 $z 的值。

示例 6.17

```
(The Script)
    print "What is your age? ";
2   chomp($age=<STDIN>);

3   $price=($age > 60 ) ? 0 : 5.55;
4   printf "You will pay \$%.2f.\n", $price;

(Output)
1   What is your age? 44
4   You will pay $5.55.

(Output)
1   What is your age? 77
4   You will pay $0.00.
```

解释：

1. 将字符串 What is your age? 打印到 STDOUT。
2. 从终端键盘读取输入，并保存到标量 $age 中，同时删除换行符。
3. 将条件运算符的结果赋值给 $price。如果 age 值大于 60，则把价格赋为问号（？）右侧的值；否则就将冒号（：）右侧的值赋给标量 $price。
4. printf 把格式化后的字符串打印到 STDOUT。

示例 6.18

```
(The Script)
1   print "What was your grade? ";
2   $grade = <STDIN>;
3   print $grade > 60 ? "Passed.\n" : "Failed.\n";

(Output)
1   What was your grade? 76
3   Passed.
(Output)
1   What was your grade? 34
3   Failed.
```

解释：

1. 要求用户提供输入。
2. 将输入值赋给标量 $grade。
3. 将条件表达式的运算结果作为参数交给 print 函数打印。如果成绩大于 60，打印 Passed，否则打印 Failed。

6.3.10 范围运算符

范围运算符即可用于标量上下文，也可用于数组上下文。在标量上下文语境中，返回值将是一个布尔值，即 1 或者 0。在数组上下文中，则会返回一个列表，其中从左到右第一个元素是范围的左边界，然后逐一递增，直到碰到范围的右边界为止。

示例 6.19

```
1   print 0 .. 10,"\n";
    0 1 2 3 4 5 6 7 8 9 10

2   @alpha=('A' .. 'Z');
print  "@alpha";'
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

3   @a=('a'..'z', 'A'..'Z');
print  "@a\n";'
    a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H
I J K L M N O P Q R S T U V W X Y Z

4   @n=( -5 .. 20 );
print "@n\n";'
    -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

解释：

1. 打印从 0 到 10 的所有整数。
2. 创建一个名为 @alpha 的数组，并把从 A 到 Z 的所有大写字母存入到该数组中。其上下文语境是数组型的。然后打印该数组内容。
3. 创建一个名为 @alpha 的数组，把所有小写字母存入一个列表，并将所有大写字母存入另一个列表。然后打印该数组内容。
4. 创建一个名为 @n 的数组，存入位于 -5 和 20 之间的所有整数。然后打印该数组内容。

6.3.11 特殊字符串运算符和函数

用户能够对字符串执行许多种操作。譬如，可以通过连接运算符将两个字符串连起来，或者通过字符串重复运算符反复连接指定数目的重复字符串。

Perl 还提供了一些支持字符串操作的特殊函数（详见表 6-14）。其中 substr 函数能从原字符串中截取指定的子串，该子串从某一偏移量开始，直到偏移量右侧的指定字符数结束。index 函数能返回指定子串在原字符串中第一次出现的偏移位置。而 length 函数则可以获取给定表达式中的字符总数。

表 6-14 字符串运算符

示 例	含 义
$str1 . $str2	连接字符串 $str1 与 $str2
$str1 x $num	将 $str1 重复 $num 次
substr($str1, $offset , $len)	在 $str1 中截取从 $offset 开始长度为 $len 的子串

（续）

示　例	含　义
index($str1, $str2)	返回 $str2 在字符串 $str1 中首次出现的位置
length(EXPR)	返回表达式 EXPR 的长度
rindex($str, $substr, POSITION)	返回 $substr 在字符串 $str 中最后一次出现的位置；如果规定了 POSITION，则从该处开始处理；否则就从字符串末尾开始处理
chr(NUMBER)	返回 ASCII 码值为 NUMBER 的字符。例如，chr(65) 将返回字母 A
lc ($str)	返回小写字符串
uc ($str)	返回大写字符串

示例 6.20

```
(The Script)
    #!/usr/bin/perl
1   $x="pop";
2   $y="corn";
3   $z="*";
4   print $z x 10, "\n";              # Print 10 stars
5   print $x . $y, "\n";              # Concatenate "pop" and "corn"
6   print $z x 10, "\n";              # Print 10 stars
7   print (($x . $y ."  ")  x  5 );   # Concatenate "pop" and "corn"
                                      # and print 5 times
8   print "\n";
9   print uc($x . $y), "!\n";         # Convert string to uppercase

(Output)
4   **********
5   popcorn
6   **********
7   popcorn popcorn popcorn popcorn popcorn
9   POPCORN!
```

解释：

1. 将标量 $x 赋值为 pop。
2. 将标量 $y 赋值为 corn。
3. 将标量 $z 赋值为 *。
4. 反复连接 10 次字符串 *，然后打印到 STDOUT。
5. 连接字符串 $x 的值 pop 与字符串 $y 的值 corn，然后打印到 STDOUT。
6. 把字符串 $z 的值 * 反复连接 10 次，然后打印到 STDOUT。
7. 将字符串 pop 和 corn 反复连接 5 次，然后打印到 STDOUT。
8. 把一个换行符打印到 STDOUT。
9. uc 函数能把字符串转换为大写形式；而 lc 函数则能把字符串转换为小写形式。

示例 6.21

```
(The Script)
1   $line="Happy New Year";
```

```
2   print substr($line, 6, 3),"\n";      # Offset starts at zero
3   print index($line, "Year"),"\n";
4   print substr($line, index($line, "Year")),"\n";
5   substr($line, 0, 0)="Fred, ";
6   print $line,"\n";
7   substr($line, 0, 1)="Ethel";
8   print $line,"\n";
9   substr($line, -1, 1)="r to you!";
10  print $line,"\n";
11  $string="I'll eat a tomato tomorrow.\n";
12  print rindex($string, tom), "\n";
(Output)
2   New
3   10
4   Year
6   Fred, Happy New Year
8   Ethelred, Happy New Year
9   Ethelred, Happy New Year to you!
12  18
```

解释：

1. 把标量 $line 赋值为 Happy New Year。
2. 把原字符串 Happy New Year 的子串 New 打印到 STDOUT。其偏移量从字节 0 开始，其子串的起始位置是第 6 位 N，持续长度是 3 个字符。返回子串 New。
3. index 函数负责返回子串在原字符串中第一次出现的位置。其中，子串 Year 是从第 10 位开始的。请注意，位偏移量必须是 0。
4. 本行把 substr 函数和 index 函数混合使用。其中，index 函数负责返回子串 Year 的开始位置。而 substr 函数则把 index 函数的返回值当作子串的开始位置。其返回的子串是 Year。
5. 把子串 Fred 插入原字符串的起始位置，即字节 0 处；并越过标量 $line 的长度 0：即字符串的开头。
6. 把 $line 的值打印到 STDOUT。
7. 把子串 Ethel 插入原字符串的起始位置，即字节 0 处；并越过标量 $line 的长度 1。
8. 将 $line 的最新值 Ethelred，Happy New Year 打印到 STDOUT。
9. 把子串 r to you ！追加到标量 $line 上，从子串的末尾开始，越过 1 个字符。
10. 将 $line 的最新值 Ethelred，Happy New Year to you ！打印到 STDOUT。
11. 给标量 $string 赋值。
12. rindex 函数负责从右向左找出第一个 tom 子串，并返回其下标位置。其结果 18 是从字符串开头的第 0 个字符开始计数的。换而言之，它是子串 tom 在 tomorrow 中的起始位置。

6.3.12 算术函数

除了算术运算符外，Perl 还提供了一系列内建的算术函数，用于计算各种算术表达式（详见表 6-15）。

表 6-15 Perl 的内建算术函数

atan2(Y,X)	返回 Y/X 的余切（arctangent）值，其范围是 -PI 到 PI
cos(EXPR)	返回表达式 EXPR（以角度单位表示）的余弦（cosine）值
cos EXPR	如果没有提供 EXPR 值，则对 $_ 求余弦值

（续）

exp(EXPR)	以 e 为底求 EXPR 的幂数
exp EXPR	如果没有提供 EXPR 值，则求 exp($_)
int(EXPR)	返回 EXPR 的整数部分
int EXPR	如果没有提供 EXPR 值，则对 $_ 求整数部分
log(EXPR)	以 e 为底求 EXPR 的对数值
log EXPR	如果没有提供 EXPR 值，则对 $_ 求对数
rand(EXPR)	返回一个介于 0 和 EXPR 之间的随机数（EXPR 必须是正数）
rand EXPR	如果没有提供 EXPR 值，则返回 0 到 1 之间的随机数
rand	请参考 srand() 函数
sin(EXPR)	返回表达式 EXPR（以角度单位表示）的正弦（sine）值
sin EXPR	如果没有提供 EXPR 值，则对 $_ 求正弦值
sqrt(EXPR)	返回 EXPR 的平方根
sqrt EXPR	如果没有提供 EXPR 值，则对 $_ 求平方根
srand(EXPR)	为求随机数操作设定种子（seed）值
srand EXPR	如果没有提供 EXPR 值，则等效于 srand（time）

此外，在一个名叫 List::Util 的包中，CPAN 还提供了一些较为少见的通用工具函数，包括：first()、max()、maxstr()、min()、minstr()、reduce()、shuffle() 以及 sum()。详见 http://perldoc.Perl.org/List/Util.html#DESCRIPTION。

生成随机数。当在 Web 上查找有关随机数生成的介绍时，读者会发现，与之相关的一个领域就是 Game > Gambling > Lotteries > Ticket Generators。赌博和彩票非常依赖随机数的生成；其他一些高级程序也依赖于随机数，譬如在 Web 上传递信息时需要使用不可预测的随机加密密钥来保证加密协议的安全性。

由程序生成的随机数又称为伪随机数。正如 Ian Goldberg 和 David Wagner 在有关 Web 安全性的著作中所阐述的，真正的随机数只有在自然界才能找到，譬如放射性元素的半衰期。除了使用外部来源，计算机还不得不自己生成这些数字。但由于计算机本身是确定性的，因此这些数字并非真的是随机的[⊖]。如果用户需要在 Perl 程序中生成随机数，可使用其内建的 rand 函数。本节后面将详细介绍该函数的用法。

rand/srand 函数。rand 函数能够返回介于 0 和 1 之间的伪随机分数。如果 EXPR 值为正数，rand 函数则会返回介于 0 和 EXPR 之间的伪随机数。srand 函数负责为 rand 函数设置随机数种子值，不过如果读者使用的是 5.004 以上版本的 Perl，则已经不再需要这个函数了。随机数种子本身也是一个随机数，Perl 会把它提供给随机数生成器，作为将要生成的新随机数的起始数字。通过为 rand 函数提供随机数种子并使用复杂的算法，程序便可产生位于某一范围内的随机数。如果为 rand 函数设

⊖ Goldberg. I , Wagner. D.,“Randomness and the Netscape Browser. How Secure is the World Wide Web?” Dr. Dobb’s Journal, http://www.ddjc.com/articles/1996/9601h/9601h.html。

定了相同的随机种子值，则它们产生的随机数序列将是相同的；而不同的随机数种子值则会产生不同的随机数序列。一般把当前时间值作为默认的随机数种子值，不过现在用户还可通过 Perl 指定更为复杂的种子。

格式：

```
rand(EXPR)
rand EXPR
rand

srand(EXPR)
srand EXPR
```

示例 6.22

```
(The Script)
    #!/usr/bin/perl
1   $num=10;
2   srand(time|$$); # Seed rand with the time or'ed to
                    # the pid of this process
3   while($num){    # srand not necessary in versions 5.004 and above
4       $lotto = int(rand(10)) + 1;
                    # Returns a random number between 1 and 10
5       print "The random number is $lotto\n";
        sleep 3;
        $num--;
    }

(Output)
5   The random number is 5
    The random number is 5
    The random number is 7
    The random number is 8
    The random number is 1
    The random number is 5
    The random number is 4
    The random number is 4
    The random number is 4
    The random number is 6
```

解释：

1. while 循环在第 7 行用到了 $num 的值，迭代执行了 10 次。
2. srand 函数把 rand 函数的随机数种子设定为惟一的值，为此将内建的 time 函数所返回内容与 Perl 程序的进程号（变量 $$）逐位进行或（OR）运算。
3. while 循环迭代执行 10 次。
4. rand 函数返回介于 1 到 10 之间的随机整数（包括 1 和 10）。然后把该值赋给 $lotto。
5. 打印随机数字的值。

示例 6.23

```
(The Script)
    #!/usr/bin/perl
1   $x=5 ;     # Starting point in a range of numbers
```

```
2   $y=15;     # Ending point

    # Formula to produce random numbers between 5 and 15 inclusive
    # $random = int(rand($y - $x + 1)) + $x;
    # $random = int(rand(15 - 5 + 1)) + 5

3   while(1){
4       print int(rand($y - $x + 1)) + $x , "\n";
5       sleep 1;
    }

(Output)
15
14
5
10
11
6
12
6
7
10
6
8
6
15
11
```

解释：

1. 标量 $x 负责指定 rand 函数生成随机数范围的起始值。
2. 标量 $y 负责指定 rand 函数生成随机数范围的末尾值。
3. 启动 while 无限循环。如要退出，用户必须按下 <Ctrl>-d（UNIX）或 <Ctrl>-z（Windows）。
4. 为 rand 函数提供一个能够产生 1 到 15 之间（包括 1 和 15）整数的公式。
5. sleep 函数使程序暂停 1 秒钟。

6.4 读者应当学到的知识

1. 操作数（operand）这个词是什么意思？
2. Perl 是如何处理表达式 "5cats"+21 的？
3. Perl 是如何处理表达式 23 . 43 的？
4. 什么是自动递增运算符？
5. 什么是伪随机数？
6. “eq”和“==”之间有什么差别？
7. “and”和“&&”含义相同吗？
8. 什么是三元运算符？
9. Perl 是如何借助关系运算符比较字符串大小的？
10. “and”和“or”之间有什么差别？
11. && 和“and”含义相同吗？

12. 相等性运算符在优先级表中的位置在哪里？
13. 相等性运算符的结合性是从左到右，还是从右到左？
14. Perl 的哪个函数能够截取字符串的一个子串？
15. 什么函数能把字符串转换为大写形式？
16. 什么运算符能够让字符串重复出现？
17. 什么运算符能够把字符串连接到一起？

6.5 下章简介

在下一章内容中，我们将介绍 Perl 的控制结构。包括：如何在 if 或 unless 结构中判断条件是否为真（true）；如何阻止语句执行；如何借助循环结构反复执行一条或多条语句；以及如何跳出循环、使用标签与嵌套循环。

练习 6

1. 打印精确到 2 位小数的三个浮点数的平均值。
2. 下面表达式的另外两种写法是什么？

```
$x = $x + 1 ;
```

3. 以缩写方式重写如下表达式：

```
$y = $y + 5 ;
```

4. 计算一个房间的大小，长、宽、高分别为 12.5、9.8、10.5 英尺。
5. 计算 15 的平方，并打印结果。
6. 下面这段程序的输出结果是什么？

```
$a = 15;
$b = 4;
$c = 25.0;
$d = 3.0;
printf ("4 + c / 4 * d = %f\n", 4 + $c / 4 * $d);
printf ("a / d * a + c = %.2f\n", $a / $d * $a + $c);
printf ("%d\n", $result = $c / 5 - 2);
printf ("%d = %d + %f\n", $result = $b + $c, $b, $c);
printf ("%d\n", $result == $d);
```

7. 给定变量值 $a=10、$b=3、$c=7、$d=20，打印 $result 的值：

```
a.$result = ( $a >= $b ) && ( $c < $d );
  print "$result\n";
b.$result = ( $a >= $b ) and ( $c < $d );
  print "$result\n";
c.$result = ( $a < $b) || ( $c <= $d );
  print "$result\n";
d.$result=( $a < $b) or ( $c <= $d );
  print "$result\n";
e.$result = $a % $b;
```

8. 编写一段名叫 convert 的程序，利用下面这个公式将华氏温度转换成摄氏温度：

   C = (F-32)/1.8

9. 创建含有如下 5 句话的数组：

   "An apple a day keeps the doctor away"
   "Procrastination is the thief of time"
   "The early bird catches the worm"
   "Handsome is as handsome does"
   "Too many cooks spoil the broth"

   希望在每次执行脚本时都随机输出其中一句话。提示：只需把数组下标设置为随机数即可。

10. 下面这个公式负责根据贷款时限和每月固定利率来计算每月应还款数。请编写一个 Perl 表达式来描述该公式的逻辑。其中：P 是预期债务总额；r 是每月利率（年利率除以 12）；n 是还款周期（对于 30 年贷款而言，n 等于 30×12 = 360），而 A 则是每月应偿还数额。

$$P = A \cdot \frac{1 - \frac{1}{(1+r)^n}}{r}$$

第 7 章　条　　件

7.1　控制结构、块与复合语句

人们通过各种各样的决定来计划日常生活，程序也是如此。图 7-1 就显示了这样一张流程图。流程图是程序内部流程的一种图形化表达形式。它可以帮助开发人员确定完成指定的任务需要作哪些选择。很多计算机专业书籍都这样写道，一种优秀的编程语言应当为用户提供如下三种程序流程控制手段：

- 顺序执行一系列语句。
- 根据判断的结果，从两条语句路径中选择一条执行。
- 不断重复执行一系列语句，直到某些条件值得到满足。

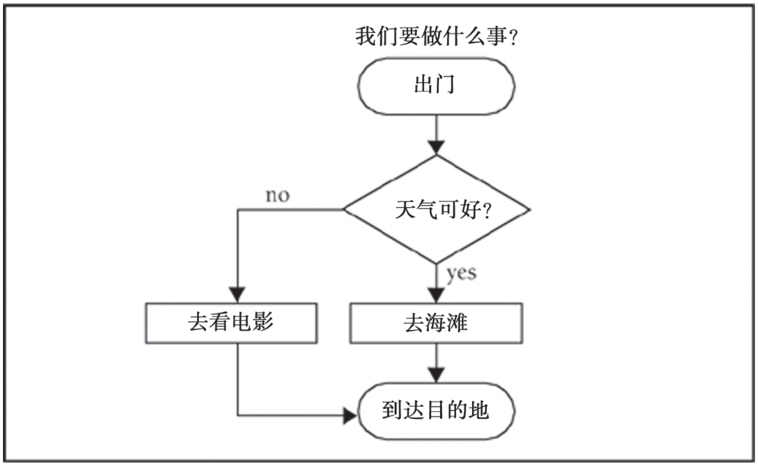


图 7-1　一张流程图

到现在为止，本书介绍的示例脚本结构都是线性的，即一条接一条地执行简单的语句。除此之外，借助分支语句和循环语句这样的控制结构，用户还能根据某些条件表达式的取值来改变程序控制流的走向。

判断结构（if、if/else、if/els if/else、unless 等）含有决定某个代码块（block）是否执行的条

件表达式。循环结构（while、until、for 或 foreach）则能让程序反复执行某个代码块，直到指定条件得以满足。

复合语句（又称为代码块）是由花括号包围起来的一组语句。代码块在语法上等效于单个语句，并常用于 if、else、while 或 for 结构的后面。C 语言并不强制要求为代码块提供花括号。但 Perl 与 C 不同，即使在 if、else、while 等之后只出现单个语句，Perl 也要求程序提供花括号。本书将在第 8 章中介绍条件修饰符，它能在单个语句中实现条件判断过程。

判断——条件结构

if 和 unless 语句。if 和 unless 语句后面一般会跟着一个位于小括号之间的条件表达式，其后则是由一个或多个语句构成的代码块。这些块都位于花括号之间。

if 语句是一种条件语句。它能够判断表达式的值，并根据判断结果做出选择。表达式都位于括号里面。与 C 语言不同，Perl 程序是在字符串上下文中为表达式取值的。如果字符串非空，则表达式的值为真；如果字符串为空，则表达式的值是假。如果表达式的值是数字，则要先把它转换成字符串，然后再判断其值。如果表达式值为真（非空），则执行下一个代码块；如果条件表达式为假（空值），Perl 就会忽略与该表达式相关联的代码块，直接跳转到脚本中的下一个可执行代码块。

unless 语句的结构与 if 语句完全相同，它只是把判断的结果反转一下。如果表达式值为假，就执行下一块；如果表达式值为真，Perl 就会忽略与该表达式相关联的代码块。

if 结构。在 if 语句中，首先出现的是其关键字 if，然后是条件表达式，其次是位于花括号里面的一个或多个语句组成的代码块。块中每个语句都以分号（;）结尾。聚合在一起的代码块往往又称作复合语句。

格式

```
if (Expression) {Block}
if (Expression) {Block} else {Block}
if (Expression) {Block} elsif (Expression)
   {Block}... else {Block}
```

示例 7.1

```
(The Script)
print "How old are you? ";
chomp($age = <STDIN>);
if ($age >= 21 ){   # If true, enter the block
        print "Let's party!\n";
}
print "You said you were $age.\n";

(Output)
1   How old are you? 32
4   Let's party!
5     You said you were 32.

-------------Run the program again -------------
(Output)
1   How old are you? 10
5   You said you were 10.
```

解释

1. 要求用户输入年龄。
2. 为标量 $age 赋值。
3. 检查标量 $age 的值。如果该值大于或等于 21（也就是说表达式值为真），则开始执行第 4 行的代码块。
4. 如果用户年龄大于 21，则打印本行内容。两边的花括号不是可有可无的，而是必须提供的！
5. 不论 if 代码块是否得以执行，程序控制流总会到达这一行语句。

if/else 结构。if 语句的另外一种形式是 if/else 结构。该结构支持双路选择。如果 if 后面第一个表达式的值为真，就执行 if 后面的语句；否则，如果 if 后面的条件表达式为假，则把控制权交给 else，来执行 else 后面的语句。else 语句不能作为独立语句而存在。它必须跟随在 if 语句之后。如果 if 语句嵌入到其他 if 语句中的话，else 语句将与前面最近的 if 语句相关联。

格式

```
if (Expression)
      {Block}
else
      {Block}
```

示例 7.2

```
(The Script)
1   print "What version of the operating system are you using? ";
2   chomp($os=<STDIN>);
3   if ($os > 2.2) {print "Most of the bugs have been worked
                           out!\n";}
4   else {print "Expect some problems.\n";}

(Output)
1   What version of the operating system are you using?  2.4
3   Most of the bugs have been worked out!

(Output)
1   What version of the operating system are you using?  2.0
4   Expect some problems.
```

解释

1. 要求用户输入。
2. 删除换行符。
3. 如果 $os 大于 2.2，则执行花括号内的代码块。
4. 如果 $os 不大于 2.2，则执行本行代码。

if/elsif/else 结构。if 语句的另一种形式则是 if/elsif/else 结构。该结构提供了多路判断的控制流。如果关键字 if 后面的第一个条件表达式值为真，就执行 if 后面的代码块。否则就判断第一个 elsif 语句。如果第一个 elsif 后面的条件表达式值为假，再判断下一个 elsif，依次类推。如果所有 elsif 语句后面的条件表达式值都为假的话，则执行 else 后面的代码块，也就是默认处理操作。

格式

```
if (Expression1)
```

```
        {Block}
elsif (Expression2)
        {Block}
elsif (Expression3)
        {Block}
else
        {Block}
```

示例 7.3

```
(The Script)
1   $hour=(localtime)[2];
2   if ($hour >= 0 && $hour < 12){print "Good-morning!\n";}
3   elsif ($hour == 12){print "Lunch time.\n";}
4   elsif ($hour > 12 && $hour < 17) {print "Siesta time.\n";}
5   else {print "Goodnight. Sweet dreams.\n";}

(Output)
4   Siesta time
```

解释

1. 把标量 $hour 赋值为当前时间。借助内置函数 localtime 返回当前小时数，也就是时间值数组的第三个元素。
2. if 语句会判断 $hour 的值是否大于或等于 0 并小于 12。如果结果是真，则执行表达式后面的代码块（即执行 print 语句）。
3. 如果第一个 if 语句值为假，就判断本行的表达式。如果 $hour 值等于 12，则执行 print 语句。
4. 如果前一个 elsif 语句值为假，而本行 elsif 语句的值为真，则执行本行的 print 语句。
5. 如果上面没有一个表达式为真，则执行默认操作，即 else 后面的语句。

unless 结构。unless 语句类似于 if 语句，只是它把 unless 后面的条件表达式判断逻辑反转了一下，即当 unless 后面条件表达式为假时，执行与之关联的代码块。

unless/else 语句和 unless/elsif 语句的执行方式等同于 if/else 与 if/elsif 语句。不过如前所述，它们也都把条件判断逻辑反转了过来。

格式

```
unless (Expression) {Block}
unless (Expression) {Block} else {Block}
unless (Expression) {Block} elsif (Expression)
   {Block}... else {Block}
```

示例 7.4

```
(The Script)
1   print "How old are you? ";
2   chomp($age = <STDIN>);
3   unless ($age <= 21 ){  # If false, enter the block
4          print "Let's party!\n";
    }
5   print "You said you were $age.\n";
(Output)
1  How old are you? 32
4  Let's party!
```

```
5  You said you were 32.

------------- Run the program again -------------

(Output)
1  How old are you? 10
5  You said you were 10.
```

解释

1. 本示例和示例 7.1 几乎一模一样，所不同的是其条件表达式逻辑被反转了过来。在判断条件表达式时，期望看到其值为假（false）而不是真（true）。本行要求用户输入年龄。
2. 为标量 $age 赋值。
3. 检查标量 $age 的值。如果该值不是大于或等于 21（也就是说表达式值为假），则开始执行第 4 行的代码块。
4. 如果用户年龄不大于 21，则打印本行内容。两边的花括号不是可有可无的，而是必须提供的！
5. 不论是否执行 if 代码块，程序控制流总会到达这一行语句。

示例 7.5

```
(The Script)
    #!/bin/perl
    # Scriptname: excluder
1   while(<>){
2       ($name, $phone)=split(/:/);
3       unless($name eq "barbara"){
            $record="$name\t$phone";
4           print "$record";
        }
    }
5   print "\n$name has moved from this district.\n";

(Output)
$ excluder names
igor chevsky    408-123-4533
paco gutierrez  510-453-2776
ephram hardy    916-235-4455
james ikeda     415-449-0066
barbara kerz    207-398-6755
jose santiago   408-876-5899
tommy savage    408-876-1725
lizzy stachelin 415-555-1234

barbara has moved from this district.
```

解释

1. 进入 while 循环。程序会把文件名作为参数提供给脚本，然后逐行读取该文件的内容。作为参数的文件名是 names。
2. 文件的每行内容都由冒号定界符隔开。程序将把其中第一个字段赋值给 $name，将第二个字段赋值给 $phone。
3. 执行 unless 语句。其逻辑是: 除非 $name 的值是 barbara，否则进入执行代码块。换而言之，只要不是 barbara，其他什么都行。
4. 打印除 barbara 之外的所有名字和电话号码。
5. 在退出循环时，打印本行信息。

7.2 循环

有时，用户需要反复执行某一个或一组语句，直到满足特定的某些条件为止。例如，需要不断向用户提相同的问题，直到收到正确答复为止；或者创建一个从 10 数到 0 的计数器；或者从头到尾修改列表中的每一项。针对此类需求，就出现了循环（loop）结构。循环能够反复执行某一段语句。Perl 的基本循环结构包括：

while

until

for

foreach

每个循环后面都跟有一个语句块，并由花括号包围起来。

7.2.1 while 循环

如果 while 之后的表达式为真，while 循环便会反复执行与之关联的代码块。如果表达式的值不是 0（非空值），它就是真的；while(1) 的值始终为真，因此会永远循环下去。如果表达式值为 0（空值），说明表达式为假；while(0) 的值始终为假，因此它永远也不会循环。

格式

```
while (Expression) {Block}
```

示例 7.6

```
(The Script)
    #!/usr/bin/perl
1   $num=0;                # Initialize $num
2   while ($num < 10){     # Test expression
    # Loop quits when expression is false or 0

3       print "$num ";
4       $num++;    # Update the loop variable $num; increment $num
5       }
6   print "\nOut of the loop.\n";

(Output)
3   0 1 2 3 4 5 6 7 8 9
6   Out of the loop.
```

解释

1. 初始化标量 $num。在进入循环前，必须先进行初始化。
2. 判断表达式的值。如果值为真，则执行花括号中的代码块。
3. 对标量 $num 执行递增操作。如果不这样做，上述表达式值将永远为真，循环也就永远不会结束。

示例 7.7

```
(The Script)
```

```
    #!/usr/bin/perl
1   $count=1;      # Initialize variables
    $beers=10;
    $remain=$beers;
    $where="on the shelf";
2   while ($count <= $beers) {
        if ($remain == 1){print "$remain bottle of beer $where ." ;}
        else {print "$remain bottles of beer $where $where .";}
        print " Take one down and pass it all around.\n";
        print "Now ", $beers - $count , " bottles of beer $where!\n";

3       $count++;
4       $remain--;
5       if ($count > 10){print "Party's over. \n";}
    }
    print "\n";
(Output)
10 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 9 bottles of beer on the shelf!
9 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 8 bottles of beer on the shelf!
8 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 7 bottles of beer on the shelf!
7 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 6 bottles of beer on the shelf!
6 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 5 bottles of beer on the shelf!
5 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 4 bottles of beer on the shelf!
4 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 3 bottles of beer on the shelf!
3 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 2 bottles of beer on the shelf!
2 bottles on the shelf on the shelf. Take one down and pass it all around.
Now 1 bottle of beer on the shelf!
1 bottle of beer on the shelf on the shelf. Take one down and pass it all around.
Now 0 bottles of beer on the shelf!
Party's over.
```

解释

1. 初始化标量 $count、$beer、$remain 和 $where。
2. 进入 while 循环，测试并判断表达式的值。
3. 对标量 $count 执行递增运算。
4. 对标量 $remain 执行递减运算。
5. 当 $count 的值大于 10 时，打印该行内容。

7.2.2　until 循环

只要 until 后面的条件表达式值为假或 0，until 结构就会反复执行代码块。只有当表达式值为真（非 0）时才退出循环。

格式

```
until (Expression) {Block}
```

示例 7.8

```
(The Script)
    #!/usr/bin/perl
1   $num=0;       # initialize
2   until ($num == 10){
        # Test expression; loop quits when expression is true or 1
3      print "$num ";
4      $num++;    # Update the loop variable $num; increment $num
5   }
6   print "\nOut of the loop.\n";

(Output)
3   0 1 2 3 4 5 6 7 8 9
6   Out of the loop.
```

解释

1. 初始化标量 $num。在进入循环前，必须先进行初始化。
2. 判断表达式的值。如果值为假，则执行花括号中的代码块。当 $num 的值等于 10 时，退出循环。
3. 对标量 $num 执行递增操作。如果不这样，上述表达式值将永远为假，循环也就永远不会结束。

示例 7.9

```
(The Script)
    #!/usr/bin/perl
1   print "Are you o.k.? ";
2   chomp($answer=<STDIN>);
3   until ($answer eq "yes"){
4       sleep(1);
5       print "Are you o.k. yet? ";
6       chomp($answer=<STDIN>);
7   }
8   print "Glad to hear it!\n";

(Output)
1   Are you o.k.? n
1   Are you o.k. yet? nope
1   Are you o.k. yet? yup
1   Are you o.k. yet? yes
8   Glad to hear it!
```

解释

1. 一开始向用户提一个问题。
2. 接收来自标准输入的用户响应，并将其保存到标量 $answer 中。然后删除换行符。
3. until 循环检查小括号里面的条件表达式，如果 $answer 的值不完全等于字符串 yes，就进入表达式后面的代码块。当 $answer 的值等于 yes 时，退出循环，跳转到第 8 行。
4. 如果 $answer 的值不等于 yes，执行这一行。换而言之，程序会暂停一分钟（sleep 1），以便在下一次提问前为用户提供考虑时间。
5. 等用户准备好后，进行下一次提问。
6. 再一次从 STDIN 读取用户的回答，并保存到 $answer 中。这一行非常重要，因为倘若 $answer 的值总是不变，那么循环便会不停执行下去。
7. 由花括号标识 until 循环代码块的结束。然后把控制权转回到第 3 行，并再次判断条件表达式的值。如果 $answer 的值为 yes，则跳转到第 8 行；否则继续重复执行代码块中的语句。
8. 在退出循环后执行本行内容。换而言之，当 $answer 的值为 yes 时才会执行本行。

do/while 与 do/until 循环。与 while、until 循环一样，do/while 与 do/until 循环也能判断条件表达式的真假值。不过，后者在至少执行完一次循环体后才会对条件表达式进行判断。

格式

```
do {Block} while (Expression);
do {Block} until (Expression);
```

示例 7.10

```
(The Script)
    #!/usr/bin/perl
1   $x = 1;
2   do {
3      print "$x ";
4      $x++;
5   } while ($x <= 10);
    print    "\n";

6   $y = 1;
7   do {
8      print "$y " ;
9      $y++;
10  } until ($y > 10);
(Output)
3   1 2 3 4 5 6 7 8 9 10
8   1 2 3 4 5 6 7 8 9 10
```

解释

1. 把标量 $x 赋值为 1。
2. 开始 do/while 循环。
3. 执行代码块。
4. 对标量 $x 执行递增操作。
5. 判断 while 后面的条件表达式值。如果为真，则在此执行代码块，依此类推。
6. 把标量 $y 赋值为 1。
7. 开始 do/until 循环。
8. 执行代码块。
9. 对标量 $y 执行递增操作。
10. 判断 until 后面的条件表达式值。如果为假，则在此执行代码块。依此类推。

7.2.3　for 循环

for 循环语句类似于 C 语言里的 for 循环。首先是关键字 for，其后是位于括号中并以分号隔开的三个表达式。用户可以省略其中任何一个或所有三个表达式，但不能省略那两个分号⊖。其中，第一个表达式负责设定变量的初始值；第二个变量用于判断循环能否继续下去；而第三个表达式则负责更新循环变量的值。

⊖　因此，无穷循环亦可写作：for(;;)。

格式

```
Expression1;
while (Expression2)
     {Block; Expression3};
```

示例 7.11

```
(The Script)
    #!/usr/bin/perl
1   for ($i=0; $i<10; $i++){   # Initialize, test, and increment $i
2       print "$i ";
    }
3   print "\nOut of the loop.\n";

(Output)
2   0 1 2 3 4 5 6 7 8 9
3   Out of the loop.
```

解释

1. for 循环含有三个表达式。第一个表达式把标量 $i 赋值为 0。该语句只执行一次。第二个表达式负责检查 $i 是否小于 10，如果是，就执行代码块（即打印 $i 的值）。最后一个表达式则负责把 $i 的值递增 1。然后再一次判断第二个表达式，并执行代码块，递增 $i。如此循环下去，直到第二个表达式判断为假。
2. 打印 $i 的值。

示例 7.12

```
(The Script)
    #!/usr/bin/perl
    # Initialization, test, and increment, decrement of
    # counters is done in one step.
1   for ($count=1, $beers=10, $remain=$beers, $where="on the shelf";
         $count <= $beers; $count++, $remain--)
       {
2   if ($remain == 1){
          print "$remain bottle of beer $where $where " ;
       }
    else {
          print "$remain bottles of beer $where $where.";
       }
       print " Take one down and pass it all around.\n";
       print "Now ", $beers - $count , " bottles of beer $where!\n";
3      if ($count == 10 ){print "Party's over.\n";}
    }
(Output)
10 bottles of beer on the shelf on the shelf. Take one down and pass it all around.
Now 9 bottles of beer on the shelf!
9 bottles of beer on the shelf on the shelf. Take one down and pass it all around.
Now 8 bottles of beer on the shelf!
8 bottles of beer on the shelf on the shelf. Take one down and pass it all around.
Now 7 bottles of beer on the shelf!

  < continues >

2 bottles of beer on the shelf on the shelf. Take one down and pass it all around.
```

```
Now 1 bottle of beer on the shelf!
1 bottle of beer on the shelf on the shelf. Take one down and pass it all around.
Now 0 bottles of beer on the shelf!
Party's over.
```

解释

1. 所有标量的初始化工作都是在 for 循环的第一个表达式中完成的。其每个初始值都由逗号分开，整个表达式则以分号结尾。在开始循环时，第一个表达式只执行一次，然后便判断第二个表达式。如果第二个表达式值为真，则执行循环体代码块。在执行代码块中所有内容后，执行第三个表达式，然后再把控制权转交给第二个表达式。如此反复。
2. 如果 for 循环的第二个表达式值为真，则执行循环体代码块。
3. 判断该语句，如果其值为真，则执行该语句，并把控制权转移到 for 循环的第二个表达式处，并递增一次 $count。

7.2.4　foreach 循环

如果读者熟悉 C shell 编程的话，就会发现 Perl 中的 foreach 循环不论在外观上还是行为上都和 C shell 的 foreach 循环极其相似。不过，这种相似性只是一种表象，这两种结构之间还是存在着很大不同的。

foreach 循环能对括号中列表（数组）里的每个元素进行迭代，并为数组中的每个元素逐一赋值，直到列表的末尾。

VARIABLE 是 foreach 代码块的一个局部变量，负责在退出循环时恢复其原先的值。反过来讲，对 VARIABLE 变量的任何改变又会影响到数组中的个别元素。如果代码块中没有出现 VARIABLE 变量的话，程序则会隐式地使用特殊变量 $_。

格式

```
foreach VARIABLE (ARRAY)
{BLOCK}
```

示例 7.13

```
(The Script)
    #!/usr/bin/perl
1   foreach $pal ('Tom', 'Dick', 'Harry', 'Pete') {
2       print "Hi $pal!\n";
    }

(Output)
2   Hi Tom!
    Hi Dick!
    Hi Harry!
    Hi Pete!
```

解释

1. foreach 关键字后面跟随着标量 $pal 和一个名字列表。其中，$pal 指向列表中从 Tom 开始的每一个名字。读者可以把 $pal 看作列表各项的别名或者引用。每次进入循环时，$pal 便指向列表中的下一项，并获取该项的值。因此，它能在 Tom 之后取 Dick，然后是 Harry，直到到达列表末尾才退出循环。
2. 每次循环都会打印 $pal 所引用的值（详见图 7-2）。

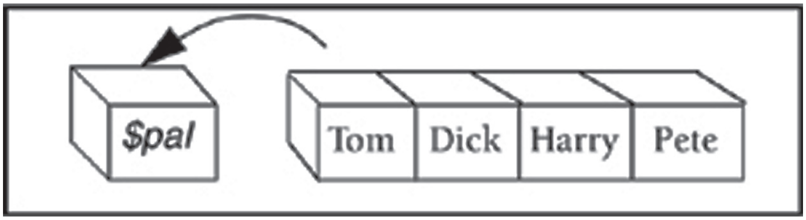


图 7-2　foreach 循环

示例 7.14

```
(The Script)
1   foreach $hour (1 .. 24){   # The range operator is used here
2     if ($hour > 0 && $hour < 12) {print "Good-morning.\n";}
3     elsif ($hour == 12) {print "Happy Lunch.\n";}
4     elsif ($hour > 12 && $hour < 17) {print "Good afternoon.\n";}
5     else {print "Good-night.\n";}
    }
(Output)
2   Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
    Good-morning.
3   Happy Lunch.
4   Good afternoon.
    Good afternoon.
    Good afternoon.
    Good afternoon.
5   Good-night.
    Good-night.
    Good-night.
    Good-night.
    Good-night.
    Good-night.
    Good-night.
    Good-night.
```

解释

1. 列表（1..24）表示取值范围是 1 到 24 的项目列表。标量 $hour 负责依次引用列表中的每一个值。本行将执行代码块，并把列表中的下一项赋值给 $hour。依此类推。
2. 检查标量 $hour 的值。如果其值大于 0 且小于 12，则执行 print 函数。
3. 如果上一个 elsif 语句值为假，则检查本语句的值。如果标量 $hour 的值等于 12，则执行本行的 print 语句。
4. 如果上一个 elsif 语句值为假，则检查本语句的值。如果标量 $hour 的值大于 12 且小于 17，则执行本行的 print 语句。
5. 如果前面的语句都为假，则执行 else 或其他默认语句。

示例 7.15

```
(The Script)
    #!/usr/bin/perl
1   $str="hello";
2   @numbers = (1, 3, 5, 7, 9);
3   print "The scalar \$str is initially $str.\n";
4   print "The array \@numbers is initially @numbers.\n";
5   foreach $str (@numbers ){
6       $str+=5;
7       print "$str\n";
8   }
9   print "Out of the loop--\$str is $str.\n";
10  print "Out of the loop--The array \@numbers is now @numbers.\n";

(Output)
3   The scalar $str is initially hello.
4   The array @numbers is initially 1 3 5 7 9.
7   6
    8
    10
    12
    14
9   Out of the loop--$str is hello.
10  Out of the loop--The array @numbers is now 6 8 10 12 14.
```

解释

1. 把标量 $str 赋值为字符串 hello。
2. 把数组 @numbers 赋值为数字型列表：1、3、5、7 和 9。
3. print 函数将 $str 的初始值打印到 STDOUT。
4. print 函数将 @numbers 的初始值打印到 STDOUT。
5. foreach 语句负责依次将列表中的每一项赋值给 $str。$str 变量在循环里是一个局部变量，负责引用列表中的每一项。因此，对 $str 的赋值操作将会影响数组 @numbers 的内容。在退出循环时，恢复其初始值。
6. 在每次循环中，将 $str 引用的值递增 5。
7. print 函数负责把 $str 的新值打印到 STDOUT。
8. 退出循环后，把 $str 的初始值打印到 STDOUT。
9. 退出循环后，把数组 @numbers 的新值与初始值一起打印到 STDOUT。

示例 7.16

```
(The Script)
    #!/usr/bin/perl
1   @colors=(red, green, blue, brown);
2   foreach (@colors) {
3       print "$_ ";
4       $_="YUCKY";
    }
5   print "\n@colors\n";

(Output)
3   red green blue brown
5   YUCKY  YUCKY  YUCKY  YUCKY
```

解释

1. 对数组 @colors 进行初始化。
2. 在 foreach 循环后面没有显式的变量，但有一个列表。由于没有显式变量，因此程序会隐式地使用特殊标量 $_。
3. 特殊变量 $_ 是实际指向当前列表项的引用。$_ 变量负责引用列表 @colors 中的每一项内容，并将其值打印到 STDOUT。
4. 把 $_ 变量赋值为字符串 YUCKY。数组 @colors 中的每一项初始值都将由 YUCKY 依次替换。
5. 本行实际改变了 @colors 数组的内容。其中，$_ 变量在进入循环前的值为空值。

7.2.5 循环控制

为了中断循环内正常执行的控制流，Perl 提供了一些循环控制标记和简单的控制语句。这些标记和语句能在条件满足时控制循环，即，将控制流直接跳转到循环底部或顶部，或者跳过控制语句条件之后的任何语句。

标记（label）。标记在循环中是可有可无的，但它能用于控制循环流。标记本身不能做任何事情。它必须与后面介绍的循环控制修饰符配合使用。不论是否出现标记，循环代码块都等价于只执行一次的循环体。如果将标记全部大写，就不会与其他保留字相混淆了。

格式

```
LABEL: while (Expression){Block}
LABEL: while (Expression) {Block} continue{Block}
LABEL: for (Expression; Expression; Expression)
   {BLOCK}
LABEL: foreach Variable (Array){Block}
LABEL: {Block} continue {Block}
```

To control the flow of loops, the following simple statements may be used within the block:

```
next
next LABEL
last
last LABEL
redo
redo LABEL
goto LABEL
```

next 语句能跳过循环体中的其他语句，直接进入下一次循环，并重新判断循环表达式的值。其效果类似于 C 语言、awk 命令和 shell 脚本中的 continue 语句。由于代码块等效于只执行一次的循环体，因此 next 语句亦可用于退出代码块，或者在 continue 代码块存在的时候跳转到 continue 代码块。

last 语句则负责退出或者中断循环。其效果类似于 C 语言、awk 命令和 shell 脚本中的 break 语句。由于代码块等效于只执行一次的循环体，因此 last 语句亦可用于退出代码块。

redo 语句能够重新执行一次循环体，同时不去判断循环表达式的值。

continue 代码块是在下一次判断循环表达式值之前需要执行的代码块。

尽管大多数程序员不建议使用 goto，但 Perl 还是提供了 goto 语句。它以标记为参数，在程序执行到 goto 语句时，会自动跳转到标记标注的语句。标记可以放在程序的任何位置，但在 do 语句或子程序中是无效的。

非循环标记块。代码块等效于只执行一次的循环体。它也是可以标记的。

redo 语句能把控制流跳转到最内层循环的顶端，或者跳到标记块（如果有标记，则类似于 goto 语句）的起始位置，并不再判断循环表达式。

示例 7.17

```
(The Script)
    #!//usr/bin/perl
    # Program that uses a label without a loop and the redo statement
1   ATTEMPT: {
2   print "Are you a great person? ";
        chomp($answer = <STDIN>);
3       unless ($answer eq "yes"){redo ATTEMPT ;}
    }

(Output)
2   Are you a great person? Nope
2   Are you a great person? Sometimes
2   Are you a great person? yes
```

解释

1. 标记是由用户定义的，位于代码块前方。其效果正如把代码块命名为 ATTEMPT。
2. 要求用户提供输入。
3. 如果 $answer 的值不是 yes，redo 语句负责重新执行代码块，此时其效果类似于 goto 语句。

示例 7.18

```
(The Script)
#!/usr/bin/perl
1    while(1){  # start an infinite loop

2    print "What was your grade?";
     $grade = <STDIN>;

3    if ($grade <0 || $grade >100) {
        print "Illegal choice\n";
        next; }     # start control at the beginning of
                    # the innermost loop
5    if  ($grade  > 89 && $grade < 101) {print  "A\n";}
     elsif  ($grade  > 79 && $grade < 90) {print  "B\n";}
     elsif  ($grade  > 69 && $grade < 80) {print  "C\n";}
     elsif  ($grade  > 59 && $grade < 70) {print  "D\n";}
     elsif  (print "You Failed."}
6    print "DO you want to enter another grade?  (y/n)";
     chomp ($choice = <STDIN>);
7    if ($choice ne "y"){last ;} # break out of the innermost
                                 # loop if the condition is true
     }

(Output)
2    What was you grade? 94
     A
6    Do you want to enter another grade (y/n)?y
2    What was your grade? 66
     D
6    Do you want to enter another grade (y/n) n
```

解释

1. 开始无穷循环。
2. 要求用户提供输入。
3. 进行逻辑判断，看 $grade 的值是否小于 0 或者大于 100。
4. 如果表达式值为假，则把控制流再次跳转到 while 循环的起始位置。
5. 判断每个 if 条件语句值。
6. 要求用户提供输入。
7. 如果条件修饰符为真，则中断最内层循环。

示例 7.19

```
(The  Script)
1   ATTEMPT:{
2       print  "What is the course number? ";
        chomp($number = <STDIN>);
        print "What is the course name? ";
        chomp($course = <STDIN>);

3       $department{$number} = $course;

        print "\nReady to quit? ";
        chomp($answer = <STDIN>);
        $answer=lc($answer);  # Convert to lowercase
4       if  ($answer eq  "yes" or $answer eq "y") {last;}
5       redo ATTEMPT;
    }
6   print  "Program continues here.\n";

(Output)
2   What is the course number? 101
    What is the course name?  CIS342
3   Ready to quit? n
2   What is the course number? 201
    What is the course name? BIO211
3   Ready to quit? n
2   What is the course number? 301
    What is the course name? ENG120
3   Ready to quit? yes
6   Program continues here.
```

解释

1. 在代码块前面添加标记 ATTEMPT。非循环代码块等效于只执行一次的循环代码块。
2. 脚本读取用户输入，并填充相关数组。这里的键和值都由用户提供。
3. 为散列变量 %department 赋值。
4. 如果用户需要退出，则由 last 语句负责把控制流跳转出代码块。
5. redo 语句负责把控制流跳回到带标记代码块的顶部，并再次执行每一条语句。
6. 在退出代码块（第 4 行）后，从本行开始继续执行程序。

嵌套的循环和标记。出现在循环体中的循环称作嵌套循环(nested loop)。程序首先初始化并判断外层循环条件，然后当完成内层循环后，再重新启动外层循环。内层循环的执行速度要快于外层循环。循环可以嵌套得非常深，但有时还需要在满足某个条件时终止其循环。一般而言，如果使用

诸如 last 和 next 这样的循环控制语句，那么仅能控制最内层循环的跳转。但有时还需要将控制流从内层循环直接跳转到外层循环，这就需要借助标记来完成了。

通过为循环语句加上标记，便可使用 last、next 和 redo 等循环控制语句来控制程序的控制流。为循环加标记就等同于给这个循环起了一个名字。

示例 7.20

```
(A Demo Script)
1  OUT:while(1){
2     < Program continues here >
3     MID: while(1){
4         if (<expression is true>) {last OUT;}
          < Program continues here >
5         INNER: while(1){
6              if (<expression is true>) {next OUT;}
               <Program continues here>
          }

      }
  }
7  print "Out of all loops.\n";
```

解释

1. 在需要的情况下，可通过本行的 OUT 标记来控制无穷 while 循环。该标记的后面是冒号和循环语句。
2. 本行继续执行程序代码。
3. 在需要的情况下，可通过本行的 MID 标记来控制内层 while 循环。
4. 倘若条件表达式值为真，则执行 last 循环控制语句，在标记 OUT 处中断该循环，并转向第 7 行。
5. 把最内层 while 循环标记为 INNER。
6. 带有 OUT 标记的 next 语句这次会把循环控制流跳转回第 1 行。
7. 这是所有循环之外的语句。如果给出了 OUT 标记，本行则是 last 语句的一个分支。

示例 7.21

```
(The Script)
1   for ($rows=5; $rows>=1; $rows--){
2      for  ($columns=1; $columns<=$rows; $columns++){
3          printf "*";
4      }
5      print "\n";
6   }
(Output)
3   *****
    ****
    ***
    **
    *
```

解释

1. 外层循环中的第一个表达式负责将标量 $rows 初始化为 5，并测试该变量值。由于其值大于等于 1，因此启动内层循环。
2. 内层循环中的第一个表达式负责将标量 $columns 初始化为 1，并测试该变量值。然后由内层循环反复完成所有操作。当内层循环执行完毕时，外层循环将从上次离开的地方继续执行，

即递减 $rows，然后再判断其值。如果其值为真，则再一次执行内层循环块，依此类推。
3. 该语句属于内层的 for 循环，每次循环都要执行到这里。
4. 内层 for 循环位于花括号之间。
5. 外层循环每次都会执行 print 语句。
6. 外层循环位于花括号中。

如果省略了标记，则循环控制语句 next、last 和 redo 都将引用最内层的循环。如果是从内层嵌套循环跳转到外层循环的话，则必须在循环语句之前提供标记。

示例 7.22

```
(The Script)
    # This script prints the average salary of employees
    # earning over $50,000 annually
    # There are 5 employees. If the salary falls below $50,000
    # it is not included in the tally

1   EMPLOYEE: for ($emp=1,$number=0; $emp <= 5; $emp++){
2       do { print "What is the monthly rate for employee #$emp? ";
                print "(Type q to quit) ";
3               chomp($monthly=<STDIN>);
4               last EMPLOYEE if $monthly eq 'q';
5               next EMPLOYEE if (($year=$monthly * 12.00) <= 50000);
6               $number++;
7               $total_sal += $year;
                next EMPLOYEE;
8       } while($monthly ne 'q');
    }
9   unless($number == 0){
10      $average = $total_sal/$number;
11      print "There were $number employees who earned over \$50,000
              annually.\n";
        printf "Their average annual salary is \$%.2f.\n", $average;
        }
    else{
        print "None of the employees made over \$50,000\n";
    }

(Output)
2   What is the monthly rate for employee #1? (Type q to quit) 4000
2   What is the monthly rate for employee #2? (Type q to quit) 5500
2   What is the monthly rate for employee #3? (Type q to quit) 6000
2   What is the monthly rate for employee #4? (Type q to quit) 3400
2   What is the monthly rate for employee #5? (Type q to quit) 4500
11  There were 3 employees who earned over $50,000 annually.
    Their average annual salary is $64000.00.
```

解释

1. 在外层循环前面添加标记 EMPLOYEE。该循环负责跟踪 5 个员工。
2. 进入 do/while 循环。
3. 脚本要求用户输入每月薪水数目。
4. 如果用户输入的是 q，则执行 last 语句，跳转到标记为 EMPLOYEE 的外层循环底部。

5. 如果条件值为真，则执行 next 语句，把控制流跳转到标记为 EMPLOYEE 的外层 for 循环的顶部。
6. 对标量 $number 执行递增运算。
7. 计算标量 $total_sal 的值。
8. 执行 next 语句，把控制流跳转到标记为 EMPLOYEE 的外层 for 循环的顶部。
9. 如果 $number 的值等于 0，说明没有人的收入超过 $50 000，因此进入该代码块执行。
10. 计算平均年收入。
11. 显示结果。

continue 代码块。位于 while 循环后面的 continue 代码块提供了类似 for 循环的变量递增效果，即便在使用 next 语句时也能发挥作用。

示例 7.23

```
(The Script)
    #! /usr/bin/perl#
    # Example using the continue block
    for ($i=1; $i<=10; $i++) {       # $i is incremented only once
        if ($i==5){
           print "\$i == $i\n";
           next;
        }
        print "$i ";
    }

    print "\n"; print '=' x 35; print "\n";
# ---------------------------------------------------------

    $i=1;
    while ($i <= 10){
        if ($i==5){
           print "\$i == $i\n";
           $i++;    # $i must be incremented here or an
                    # infinite loop will start
           next;
        }
        print "$i ";
        $i++;           # $i is incremented again
    }

    print "\n"; print '=' x 35; print "\n";
    # -------------------------------------------------------
    # The continue block allows the while loop to act like a for loop
    $i=1;
    while ($i <= 10) {
        if ($i == 5) {
            print "\$i == $i\n";
            next;
        }
        print "$i ";
    }continue {$i++;}    # $i is incremented only once

(Output)
1 2 3 4 $i == 5
```

```
6 7 8 9 10
===================================
1 2 3 4 $i == 5
6 7 8 9 10
===================================
1 2 3 4 $i == 5
6 7 8 9 10
```

解释

1. 进入 for 循环并循环 10 次。
2. 如果 $i 的值为 5，进入代码块，并……
3. ……打印 $i 的值。
4. next 语句将把控制权返回给 for 循环。在返回到 for 循环时，总是在判断第二个表达式之前首先判断其第三个表达式。然后在测试第二个表达式之前将 $i 递增。
5. 在每次循环内打印 $i 的值。直到 $i 等于 5 为止。
6. 将 $i 初始化为 5。
7. 如果表达式值为真，则进入 while 循环体。
8. 如果 $i 值等于 5，则显示 $i 的值。
9. $i 递增 1。如果在这里不递增它，该变量就永远不会增加了，因此该循环也就成了死循环。
10. next 语句把控制流再一次跳转到 while 循环的起始位置，并再次判断 while 后面的条件表达式。
11. 显示 $i 的当前值。
12. 在递增 $i 之后，把控制流跳转回 while 循环的起始位置，并再次判断 while 后面的条件表达式。
13. 当 $i 小于等于 10 时，进入循环体。
14. 如果 $i 等于 5，则进入该代码块。
15. 显示 $i 的当前值。
16. next 语句一般会使控制流跳转到 while 循环的起始位置。但在这里，由于循环尾部出现了 continue 代码块，因此控制流首先会进入 continue 代码块，然后才跳转到 while 循环的起始位置，并判断其条件表达式。
17. 显示 $i 的当前值。
18. 在 next 语句把控制流跳转到 while 循环顶部之前，首先执行 while 循环体尾体部的 continue 代码块。如果没有 next 语句，程序则会在循环体的最后一个语句之后执行 continue 代码块。

7.2.6 switch 语句

switch 语句是另一种类型的控制语句，类似于 if/elsif/else 语句。所不同的是，它是通过检查表达式与一组 case 标记语句值的匹配情况来判断控制条件的。当发现一个匹配时，便把程序控制权转交给那个匹配的标签所对应的代码块。下面这个示例演示了在 C 语言、PHP 等中是如何通过 switch 语句来设计程序的。

```
switch (expression) {
  case value1 :
    /* statements */
    break;
  case value2 :
    /* statements */
```

```
    break;
  case value3 :
    /* statements */
    break;
  default:
    /* statements */
    break;
}
```

尽管 switch/case 机制是大多数语言都已提供的常用控制结构，但 Perl 5 却不支持它（它将出现在 Perl 6 中）。在 Perl 中，读者可以借助传统的 if/elsif/else 结构或利用标签和 next 语句来达成相同的效果。由于代码块（不管它前面有没有标签）等效于只执行一次的循环，而诸如 last、next 和 redo 之类的循环控制语句可以用在上述代码块内部，因此读者可以通过添加 do 代码块并在其中执行一系列命令的方式间接实现名为“phoney”的 switch 语句。详见示例 7.24。

示例 7.24

```
(The Script)
    #! /usr/bin/perl
    $hour=0;
    while($hour < 24) {
      SWITCH: {      # SWITCH is just a user-defined label
          $hour < 12                 && do {   print "Good-morning!\n";
                                               last SWITCH;};

          $hour == 12                && do {   print "Lunch!\n";
                                               last SWITCH;};

          $hour > 12 && $hour <= 17  && do {   print "Siesta time!\n";
                                               last SWITCH;};

          $hour > 17                 && do {   print "Good night.\n";
                                               last SWITCH;};
      }  # End of block labeled SWITCH

      $hour++;
    } # End of loop block
(Output)
Good-morning!
Good-morning!
Good-morning!

<Output continues>

Good-morning!
Good-morning!
Good-morning!
Lunch!
Siesta time!
Siesta time!
Siesta time!
Siesta time!
Siesta time!
Good night.
Good night.
Good night.
```

```
Good night.
Good night.
```

解释

1. 在进入循环前，将标量 $hour 赋值为初始值 0。
2. 判断 while 循环表达式的值。
3. SWITCH 标签负责对代码块进行标记。它只是一个标签，仅此而已。
4. 在进入代码块后，首先判断表达式的值。如果 $hour 值小于 12，就会执行 && 右侧的表达式内容。这里是一个 do 代码块，并将顺次执行代码块中的一系列语句。最后返回 last 语句的值。
5. label 语句将控制流跳转到 SWITCH 标签块的末尾，即第 8 行处。
6. 当前面的表达式值为 false 时，执行本行语句。当 $hour 值等于 12 时，读取本行。
7. 当前面的表达式值为 false 时，执行本行语句。当 $hour 值大于 12 且小于等于 17 时……
8. 如果本行表达式为 true，则执行 do 代码块内容。当 $hour 值大于 17 时执行本行表达式……
9. 每次完成循环后，将标量 $hour 递增 1。

Switch.pm 模块。如果读者需要使用 switch 语句，则可使用 Switch.pm 模块。该模块可从 http://search.cpan.org/~rgarcia/Switch-2.13/Switch.pm 处找到。详见示例 7.25。

示例 7.25

```
(The Script)
1 use Switch; # Loads a Perl module

2 print "What is your favorite color? ";
  chomp($color=<STDIN>);
3 switch("$color"){
4  case "red"  { print "Red hot mama!\n"; }
   case "blue" { print "I got a feeling called the blues.\n"; }
   case "green" { print "How green my valley\n";}
   case "yellow" { print "In my yellow submarine";}
5  else          { print "$color is not in our list.\n";}
 }
6   print "Execution continues here....\n";

(Output)
2 What is your favorite color? blue
  I got a feeling called the blues.
6 Execution continues here....
---------------------------------------
2 What is your favorite color? pink
  pink is not in our list.
6 Execution continues here....
```

解释

1. 本行代码负责将 Switch.pm 模块从标准 Perl 库载入程序内存空间中。
2. 要求用户输入其最喜欢的颜色，该信息将存储到标量 $color 中。
3. switch 语句负责检查变量 $color 的值，并与下面 case 语句中提供的字符串进行比较。这里的“case”亦可换成 if/elsif 结构中的 elsif 语句。
4. 如果颜色值为“red”，则执行 case“red”后面的代码块，然后跳转到第 6 行。如果不是“red”，则继续判断下一个 case 语句“blue”，依此类推。
5. 如果所有的 case 内容都不匹配 switch 表达式，则执行 else 代码块。

7.3 读者应当学到的知识

1. 什么是控制结构？
2. 什么是块？
3. if 或 else 结构之后的花括号是可选的吗？
4. else 块的功能是什么？
5. 哪种控制结构支持多重选择？
6. while 循环和 do/while 循环之间有什么不同？和 until 循环相比呢？
7. 在尚未到达代码块末尾时，应当如何跳出循环？
8. 什么是 redo 语句？它和 next 语句之间有什么不同？
9. 为什么在进行循环控制时使用 next 语句要优于用 redo ？
10. 什么是无穷循环？
11. foreach 循环是如何工作的？
12. Perl 支持 switch 语句吗？什么是 Switch.pm 模块？
13. continue 代码块的用途是什么？

7.4 下章简介

在下一章里，读者将学习如何借助正则表达式来实现模式匹配。正则表达式是 Perl 提供的最优秀也是最重要的语言特性之一。读者将学到表达式修饰符的概念，以及如何在文本字符串中找到指定模式。如果读者熟悉 UNIX 提供的 grep 或 vi 编辑器，那么就还可以了解到 Perl 是如何通过其匹配和替换运算符和大量正则表达式元字符来改进并简化模式匹配动作的。

练习 7 什么是条件？

1. 物理学家告诉我们，最低的温度是绝对零度，它等于华氏 -459.69°。
   a. 接收用户输入：起始温度、结束温度和增值（单位都是华氏度）。
   b. 检查输入错误：即小于绝对零度的情况和结束温度小于起始温度的情况。如果发现这两种情况，则让程序发送消息到 STDERR。
   c. 打印标题“Fahrenheit Celcius”。打印从开始到结束温度之间的所有值。请使用循环机制，其转换公式为：C=(F-32)/1.8。
2. 要求用户输入一系列以空格分隔开的分值，并将这些分值保存到字符串 $input 中。
   a. 拆分字符串 $input，创建一个新数组。
   b. 借助 foreach 循环获得所有分值之和。
   c. 打印平均值。
3. 编写一段脚本，从一叠纸牌中随机抽取 10 张，并打印结果值。
   a. 该脚本应当先借助 foreach 循环构建一套 52 张牌。
   b. 外层循环负责按照花色遍历整套纸牌：梅花、方块、红桃、黑桃。内层循环则针对每一种花色遍历所有数字：A、1 到 10、J、Q 以及 K。从各花色获得的纸牌都将赋值到一个数组中。
   c. 使用 rand() 函数从上述数组中随机获得一张纸牌。这样做便能保证获得的 10 张纸牌互不重复。

第 8 章　正则表达式——模式匹配

8.1　什么是正则表达式

如果读者熟悉 UNIX 实用工具，譬如 vi、sed、grep 和 awk，就应该了解用于定界搜索模式的正则表达式和元字符。在 Perl 中，它们又回来啦。

那么，到底什么是正则表达式呢？正则表达式是一种序列或字符模式，负责在搜索和替换文本时对文本内容进行字符串匹配。简单的正则表达式直接由待匹配字符串或字符集构成。正则表达式一般以斜杠（/）作为定界符[⊖]。在执行模式匹配时，Perl 的默认搜索对象是特殊变量 $_ 的内容。$_ 就像影子一样，有时可以使用，有时又不能。不过读者不必担心，在读完本章后，一切就都水落石出了。

示例 8.1

```
/abc/
?abc?
```

解释

1. 模式 abc 位于斜杠之间。如果要在字符串或文本文件中匹配该模式，则所有含有字符串 abc 的字符串都会匹配成功。
2. 模式 abc 位于问号之间。在搜索该模式时，则只有该字符串的第一个字符匹配即可（详见附录中的 reset 函数）。

8.2　表达式修饰符与简单语句

所谓简单语句，就是以分号结尾的表达式。Perl 提供了一组修饰符，负责根据某些条件进一步求表达式的值。在简单语句中可能含有以这类修饰符结尾的表达式。这些修饰符及其表达式都是以分号结尾的。在对正则表达式求值时，使用修饰符要比使用完整的条件结构（见第 7 章）显得更为简洁。

这些修饰符包括：

[⊖] 实际上，任何字符串都可以成为定界符，详见表 7.3 以及示例 7.12。

if
unless
while
until
foreach

8.2.1　条件修饰符

if 修饰符。if 修饰符用于控制含有两个表达式的简单语句。如果 Expression1 表达式为真，则执行 Expression2 表达式的内容。

格式

```
Expression2 if Expression1;
```

示例 8.2

```
(In Script)
$x = 5;
print $x if $x == 5;

(Output)
5
```

解释

1. 将 $x 赋值为 5。只有当 $x 等于 5 时，程序才会打印 $x 的值。
2. if 修饰符必须放在语句末尾。在本示例中，它负责控制 print 函数的执行。如果表达式 $x==5 值为真，则打印 $x 的值。也可将其改成为 if($x==5) {print $x;}

示例 8.3

```
(In Script)
$_ = "xabcy\n";
print if /abc/;    # Could be written: print $_ if $_ =~ /abc/;

(Output)
xabcy
```

解释

1. 将标量型变量 $_ 赋值为字符串 xabcy。
2. 如果 if 修饰符后面直接跟随着正则表达式，Perl 就会假定待匹配的字符串位于 $_，它是默认的模式匹配占位符。如果字符串中任意位置的子串与正则表达式 abc 匹配，则打印 $_ 的值 xabcy ⊖。该表达式亦可写作 if $_=~/abc/（本章最后将讨论 =~ 模式匹配运算符的用法）。

示例 8.4

```
(In Script)
$_ = "I lost my gloves in the clover.";
print "Found love in gloves!\n" if /love/;
```

⊖ $_ 也是 print 函数的默认输出内容。

```
                    # Long form: if $_ =~ /love

(Output)
Found love in gloves!
```

解释

1. 将 $_ 赋值为字符串 I lost my gloves in the clover。
2. 如果正则表达式 love 与 $_ 相匹配，则打印字符串 Found love in gloves！；否则不打印。本例中，glove 与 clover 字符串均能匹配正则表达式 love。搜索过程从字符串最左端开始，因此在到达 clover 之间就已经在 glover 处匹配为真了。如果显式地将 $_（或其他标量）用在 if 修饰符之后的话，则应当在处理正则表达式时使用＝～模式来匹配运算符。

8.2.2　DATA 文件句柄

后面的示例将使用特殊的 DATA 文件句柄作为 while 循环中的一个表达式。这样便可从执行它的同一脚本中直接获得数据,而不是从另一个文件里读取输入（详见第 10 章“获得文件句柄”）。数据本身内容保存在每个脚本末尾的特殊实量 _DATA_ 之后。_DATA_ 标志着脚本逻辑的结束，并负责打开 DATA 文件句柄以读取数据。每当读取 <DATA> 的一行内容之后，就将它赋值给特殊标量 $_。尽管在这里 $_ 是隐含的，用户也可以显式地使用它，或者用其他标量代替它。下面示例展示了它的调用格式。

格式

```
while(<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

Or you could use the $_ explicitly as follows:

```
while($_=<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

Or use another variable instead of $_ as follows:

```
while($inputline=<DATA>){
    Do something with the data here
}
_ _DATA_ _
    The actual data is stored here
```

示例 8.5

```
(The Script)
    while(<DATA>){
        print if /Norma/;      # Print the line if it matches Norma
    }
```

```
3    _ _DATA_ _
     Steve Blenheim
     Betty Boop
     Igor Chevsky
     Norma Cord
     Jon DeLoach
     Karen Evich
(Output)
Norma Cord
```

解释

1. 特殊的 DATA 文件句柄能从 _DATA_ 后面的文本中读取输入内容。当进入 while 循环时，会把输入行内容保存到标量型变量 $_ 中。在这里，保存在 $_ 中的第一行内容是 Steven Blenheim。下一轮循环则会把 Betty Boop 保存到 $_ 里。如此反复，直到读取并处理完毕 _DATA_ 之后的全部行。
2. 只打印含有正则表达式 Norma 的行。其中，$_ 是模式匹配操作的默认对象。亦可写作 print $_ if $_=~/Norma/;。
3. DATA 文件句柄能够从 _DATA_ 标记之后的行获取数据。

示例 8.6

```
(The Script)
1    while(<DATA>){
2        if /Norma/ print;     # Wrong!
     }

3    _ _DATA_ _
     Steve Blenheim
     Betty Boop
     Igor Chevsky
     Norma Cord
     Jon DeLoach
     Karen Evich

(Output)
Execution of script aborted due to compilation errors.
```

解释

1. 特殊文件句柄 DATA 能从 _DATA_ 标记后面的文本中获得输入内容。while 循环遍历文本的每一行，并将每一行内容都赋值给 $_。$_ 是默认的标量变量，用于保存行输入内容并执行模式匹配。
2. 修饰符必须位于表达式末尾，否则就会产生语法错误。该语句应当是：print if /Norma/ 或者 if(/Norma/) {print;}。

unless 修饰符。unless 修饰符用于控制由两个表达式构成的简单语句。如果 Expression1 表达式为假，则执行 Expression2 表达式内容。与 if 修饰符一样，unless 修饰符也位于语句尾部。

格式

```
Expression2 unless Expression1;
```

示例 8.7

```
(The Script)
1   $x=5;
2   print $x unless $x == 6;

(Output)
5
```

解释

unless 修饰符能控制 print 语句。如果表达式 $x==6 值为假，则打印 $x 的值。

示例 8.8

```
(The Script)
1   while(<DATA>){
2       print unless /Norma/;  # Print line if it doesn't match Norma
    }

3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Jon DeLoach
Karen Evich
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中读取输入内容。这里进入 while 循环，读取 _DATA_ 下面的第一行内容并赋值给 $_，依此类推。
2. 匹配并打印所有不含模式 Norma 的行。
3. 文件句柄 DATA 从 _DATA_ 标记之后的行读取数据。

8.2.3 循环修饰符

while 修饰符。 只要第一个表达式为真，while 循环修饰符便会重复执行第二个表达式。

格式

```
Expression2 while Expression1;
```

示例 8.9

```
(The Script)
1   $x=1;
2   print $x++,"\n" while $x != 5;

(Output)
```

```
1
2
3
4
```

解释

当 $x 不等于 5 时，Perl 打印 $x 的值。

until 修饰符。只要第一个表达式值为假，while 循环修饰符便会重复执行第二个表达式。

格式

```
Expression2 until Expression1;
```

示例 8.10

```
(The Script)
1   $x=1;
2   print $x++,"\n" until $x == 5;

(Output)
1
2
3
4
```

解释

1. 将 $x 赋初始值为 1。
2. Perl 反复打印 $x 的值，直到 $x 等于 5 为止。其中，设置 $x 初值为 1，并在每次循环结束时递增它。请读者务必小心，别陷入无穷循环。

foreach 修饰符。foreach 修饰符会逐个判断列表中每个元素的值，并通过标量 $_ 依次引用各个列表元素。

示例 8.11

```
(The Script)
1   @alpha=(a .. z, "\n");
2   print foreach @alpha;

(Output)
abcdefghijklmnopqrstuvwxyz
```

解释

1. 将数组 @alpha 赋值为一组小写字。
2. 逐个地将列表每一项赋予 $_ 变量，直到遍历完列表中所有的项。

8.3 正则表达式运算符

正则表达式运算符用于在替换和搜索时提供匹配的模式。其中，m 运算符可用于匹配模式，而 s 运算符则用于替换模式。

8.3.1 m 运算符与匹配

运算符 m 用于匹配模式。如果正则表达式两侧的定界符是正斜杠的话，m 运算符就是可有可无的；但如果用户在程序中改变了定界符，它就是必需的。当正则表达式本身含有正斜杠（譬如搜索生日 3/15/95 或路径 /usr/var/admin）时，用户就有可能改变其定界符。表 8-1 列出了常用的几种匹配修饰符。

格式

```
/Regular Expression/      default delimiter
m#Regular Expression#     optional delimiters
m{regular expression}     pair of delimiters
```

表 8-1 匹配修饰符

修饰符	含 义
i	关闭大小写敏感性
m	将字符串作为多行处理
o	只编译模式一次。用于优化搜索流程
s	嵌入换行符时，将字符串作为单行处理
x	允许在正则表达式中提供注释，并忽略空白字符
g	全局匹配。即查找所有具体值。如果用于数组型上下文语境，则会返回一个列表；如果用于标量型上下文语境，则返回真或假

示例 8.12

```
1   m/Good morning/
2   /Good evening/
3   /\/usr\/var\/adm/
4   m#/usr/var/adm#
5   m(Good evening)
6   m'$name'
```

解释

1. 本例不需要使用 m 运算符，因为其正则表达式定界符是正斜杠。
2. 以正斜杠作为定界符，因此 m 运算符是可有可无的。
3. 对搜索路径中的每个正斜杠字符都通过反斜杠来引用，这样就不会与正斜杠定界符相混淆了。
4. 这里必须使用 m 运算符，因为它用镑符号（#）代替正斜杠作为定界符。镑符号定界符能确定并简化前一个示例。
5. 如果首定界符是括号、方括号、角括号或花括号的话，则尾定界符必须是与之对应的闭字符。譬如：m(expression)、m[expression]、m<expression> 以及 m{expression}。
6. 如果定界符是单引号的话，则不允许插入变量值。换而言之，本例中的 $name 将作为一个实量来处理。

示例 8.13

```
(The Script)
1   while(<DATA>){
2       print if /Betty/;     # Print the line if it matches Betty
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Betty Boop
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 匹配并打印与模式 Betty 相匹配的所有行。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.14

```
(The Script)
1   while(<DATA>){
2       print unless /Evich/;     # Print line unless it matches Evich
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 匹配并打印与模式 Evich 相匹配的所有行。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.15

```
(The Script)
1   while(<DATA>){
2       print if m#Jon#     # Print the line if it matches Jon
```

```
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Jon DeLoach
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 这里需要使用 m 运算符，因为其正则表达式定界符不是正斜杠，而是镑符号（#）。打印与 Jon 匹配的行。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.16

```
(The Script)
1   while(<DATA>){
2       print if m(Karen E);    # Print the line if it matches Karen E
    }
3   $name="Jon";
4   $_=qq/$name is a good sport.\n/;
5   print if m'$name';
6   print if m"$name";

7   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
2   Karen Evich
5   <No output>
6   Jon is a good sport.
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 这里需要使用 m 运算符，因为其正则表达式定界符由默认的正斜杠改成了一组开闭的括号。这里可供选用的其他字符包括方括号、花括号、角括号以及单引号。如果使用的是单引号，并且正则表达式内含有变量的话，则不会解释插入该变量的值。打印与 Karen E 相配的行。
3. 将标量 $name 赋值为 Jon。
4. 将 $_ 赋值为包含标量 $name 的字符串。

5. 当定界符是一组单引号时，不解释正则表达式中的变量值。由于这里在 $_ 中没有找到 $name 的值，因此不会打印其内容。
6. 如果正则表达式是位于双引号中的话，则会插入 $name 变量的值。如果含有 Jon，就打印 $_ 中的字符串。
7. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

g 修饰符：全局匹配。修饰符 g 用于产生全局匹配效果，即匹配行中所有模式的具体值。如果不使用 g 的话，模式只会与碰到的第一个值相匹配。m 运算符将返回匹配模式列表。

格式

```
m/search pattern/g
```

示例 8.17

```
(The Script)
    #!/usr/bin/perl
1   $_ = "I lost my gloves in the clover, Love.";
2   @list=/love/g;
3   print "@list.\n";

(Output)
3   love love.
```

解释

1. 将标量 $_ 赋值为文本字符串。
2. 如果使用修饰符 g 进行搜索的话，把匹配项存储到数组 @list 中。本例会在字符串中匹配成功两次，第一次是在 glove 中，第二次是在 clover 里。Love 是不匹配的，因为 L 是大写字母。
3. 打印匹配的项列表。

i 修饰符：大小写不敏感。 当 Perl 执行模式匹配时，模式是对大小写敏感的。如果用户需要关闭大小写敏感性，则应当在匹配运算符的最后一个定界符后面加上 i 修饰符（即 insensitive）。

格式

```
m/search pattern/i
```

示例 8.18

```
1   $_ = "I lost my gloves in the clover, Love.";
2   @list=/love/gi;
3   print "@list.\n";

(Output)
3   love love Love.
```

解释

1. 将标量 $_ 赋值为文本字符串。
2. 这一次使用 i 修饰符关闭大小写敏感性。此时，love 和 Love 将都能匹配，并赋值给数组 @list。
3. 找到三次匹配模式，打印匹配项列表。

保存模式的特殊标量。特殊标量 $& 中保存着上一次成功的搜索过程中所匹配的字符串值。&`

保存了成功匹配模式之前所找到的内容。&' 则负责保存成功匹配模式之后找到的内容。

示例 8.19

```
1    $_="San Francisco to Hong Kong\n";

2    /Francisco/;      # Save 'Francisco' in $& if it is found
3    print $&,"\n";

4    /to/;
5    print $`,"\n";    # Save what comes before the string 'to'

6    /to\s/;           # \s represents a space
7    print $', "\n";   # Save what comes after the string 'to'

(Output)
3    Francisco
5    San Francisco
7    Hong Kong
```

解释

1. 将标量 $_ 赋值为文本字符串。
2. 搜索模式含有正则表达式 Francisco。Perl 会在 $_ 变量中搜索该模式。如果找到的话，就将模式 Francisco 保存到另一个特殊标量 $& 中。
3. 程序匹配模式 Francisco，将其保存到特殊标量 $& 中并打印。
4. 搜索模式含有正则表达式 to。Perl 会在 $_ 变量中搜索该模式。如果与它匹配的话，就把模式左侧的字符串 San Francisco 保存到特殊标量 $` 中（注意，这里是反引号）。
5. 打印 $` 的值。
6. 搜索模式含有正则表达式 to\s（to 后面跟着空格，其中 \s 代表空格）。Perl 会在 $_ 变量中搜索该模式。如果与它匹配的话，就把模式右侧的字符串 Hong Kong 保存到特殊标量 $' 中（注意，这里是单引号）。
7. 打印 $' 的值。

x 修饰符: 表达修饰符。修饰符 x 用于在正则表达式中放入注释或者空白字符（空格、制表符、换行符等），以便让表达式含义更明确。而这些字符是不作为正则表达式的组成部分来解释的。因此可以通过它在正则表达式中描述其意图。

示例 8.20

```
1    $_="San Francisco to Hong Kong\n";
2    /Francisco  # Searching for Francisco
     /x;
3    print "Comments and spaces were removed and \$& is $&\n";

(Output)
3    Comments and spaces were removed and $& is Francisco
```

解释

1. 将标量 $_ 赋值为文本字符串。
2. 搜索模式由 Francisco 及其后面的空格、注释以及另一个空格符组成。修饰符 x 允许将附加

的空白字符和注释插入到模式中，而不是作为搜索模式的一部分去解释。

3. 打印出的内容表明，整个搜索过程没有受到注释与空白字符的影响。$& 中含有搜索结果匹配项。

8.3.2 s 运算符与替换

运算符 s 用于替换操作。替换运算符负责用第二个正则表达式替换第一个正则表达式。它也可以变换不同的定界符。位于最后一个定界符之后的 g 修饰符表明这是一次全局修改操作。s 运算符返回的结果是其完成的替换数目。如果不使用它的话，只有模式的第一个具体值能受到替换操作的影响。表 8-2 列出了常见的几种替换修饰符。

特殊内建变量 $& 能获得字符串搜索过程中找到的任何内容。

格式

```
s/old/new/;
s/old/new/i;
s/old/new/g;
s+old+new+g;
s(old)/new/;   s[old]{new};
s/old/expression to be evaluated/e;
s/old/new/ige;
s/old/new/x;
```

示例 8.21

```
s/Igor/Boris/;
s/Igor/Boris/g;
s/norma/Jane/i;
s!Jon!Susan!;
s{Jon} <Susan>;
s/$sal/$sal * 1.1/e
s/dec/"Dec" . "ember"     # Replace "dec" or "Dec" with "December"
 /eigx;
```

表 8-2 替换修饰符

修饰符	涵 义
e	将替换一侧作为表达式来求值
i	关闭大小写敏感性
m	将字符串作为多行处理
o	只编译模式一次。用于优化搜索流程
s	嵌入换行符时，将字符串作为单行处理 [a]
x	允许在正则表达式中提供注释，并忽略空白字符
g	全局匹配。即查找所有具体值

a. 其中，m、s 和 x 选项只定义于 Perl5 中。

示例 8.22

```
(The Script)
1   while(<DATA>){
2       s/Norma/Jane/;      # Substitute Norma with Jane
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Jane Cord
Jon DeLoach
Karen Evich
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 对于 $_ 变量中含有 Norma 正则表达式的行，替换运算符 s 会把每一行上 Norma 的第一个具体值替换为 Jane。
3. 无论是否发生替换，打印每一行内容。
4. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.23

```
(The Script)
1   while($_= <DATA>){
2        print if s/Igor/Ivan/;      # Substitute Igor with Ivan
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Ivan Chevsky
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 对于 $_ 变量中含有 Norma 正则表达式的行，替换运算符 s 会把每一行上 Igor 的第一个具体值替换为 Ivan。只有在替换成功时才会打印该行内容。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

改变替换定界符。一般而言，搜索模式和替换字符串两边都应当是正斜杠定界符。而在 s 运算符后，任何非字母数字的字符都可用于代替正斜杠成为定界符。例如，如果 s 运算符后面有 #，它就是用于替换模式的定界符。如果使用一对小括号、花括号、方括号或角括号作为搜索模式的定界符的话，任何其他类型的定界符都可以用于替换模式，譬如 s(John)/Joe/。

示例 8.24

```
(The Script)
1   while(<DATA>){
2       s#Igor#Boris#;        # Substitute Igor with Boris
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
Betty Boop
Boris Chevsky
Norma Cord
Jon DeLoach
Karen Evich
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 运算符 s 后面的定界符改成了镑符号（#）。本行三个定界符都是镑符号。本例将以 Boris 替换正则表达式中的 Igor。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.25

```
(The Script)
1   while(<DATA>){
2       s(Blenheim){Dobbins};      # Substitute Blenheim with Dobbins
3       print;
    }
4   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Dobbins
Betty Boop
```

```
Igor Chevsky
Norma Cord
Jon DeLoach
Karen Evich
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 使用括号作为定界符搜索模式 Blenhein；使用正斜杠作为定界符替换模式 Dobbins。
3. 打印输出，其内容表明发生了替换操作，成功地将 Blenhein 替换为 Dobbins。
4. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

g 修饰符：全局替换。修饰符 g 用于产生全局替换效果，即替换行中所有模式的具体值。如果不使用 g 的话，则每行只会替换模式的第一个具体值。

格式

```
s/search pattern/replacement string/g;
```

示例 8.26

```
(The Script)
# Without the g option
    (The Script)
1   while(<DATA>){
2       print if s/Tom/Christian/;  # First occurrence of Tom on each
                                    # line is replaced with Christian
    }
3   _ _DATA_ _
    Tom Dave Dan Tom
    Betty Tom Henry Tom
    Igor Norma Tom Tom

(Output)
Christian Dave Dan Tom
Betty Christian Henry Tom
Igor Norma Christian Tom
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 将读取的每一行中匹配 tom 的第一个值替换为 Christian。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.27

```
(The Script)
# With the g option
1   while(<DATA>){
2       print if s/Tom/Christian/g;  # All occurrences of Tom on each
                                     # line are replaced with Christian
    }
3   _ _DATA_ _
```

```
        Tom Dave Dan Tom
        Betty Tom Henry Tom
        Igor Norma Tom Tom

    (Output)
    Christian Dave Dan Christian
    Betty Christian Dick Christian
    Igor Norma Christian Christian
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 由于这里使用了 g 选项，因此替换操作是全局的。本例将读取的每一行中匹配 tom 的所有值全部替换为 Christian。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

i 修饰符：大小写不敏感。当 Perl 执行模式匹配时，模式是对大小写敏感的。如果用户需要关闭大小写敏感性，则应当在匹配运算符的最后一个定界符后面加上 i 修饰符（即 insensitive）。

格式

```
s/search pattern/replacement string/i;
```

示例 8.28

```
(The Script)
    # Matching with the i option
1   while(<DATA>){
2      print if /norma cord/i;     # Turn off case sensitivity
    }
3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Norma Cord
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 如果不使用 i 选项，则正则表达式 /norma cord/ 将没有匹配项，因为读入的行中存在大写字母。i 选项则能关闭大小写敏感性。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.29

```
(The Script)
1   while(<DATA>){
2       print if s/igor/Daniel/i;    # Substitute igor with Daniel
```

```
    }

3   _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Daniel Chevsky
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。
2. 如果指定了 i 选项，则在替换操作中，正则表达式就是大小写不敏感的。如果匹配到了 igor 或 Igor（或其他任何大小写混合的匹配情况），则将其替换为 Daniel。
3. 文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

e 修饰符：求表达式值。在进行替换操作时，可能需要去求表达式或者函数的值。本修饰符负责将搜索端内容替换成上述求值的结果。

格式

```
s/search pattern/replacement string/e;
```

示例 8.30

```
(The Script)
    # The e and g modifiers
1   while(<DATA>){
2       s/6/6 * 7.3/eg;       # Substitute 6 with product of 6 * 7.3

3       print;
    }
    _ _DATA_ _
    Steve Blenheim   5
    Betty Boop       4
    Igor Chevsky     6
    Norma Cord       1
    Jon DeLoach      3
    Karen Evich      66


(Output)
Steve Blenheim     5
Betty Boop         4
Igor Chevsky       43.8
Norma Cord         1
Jon DeLoach        3
Karen Evich        43.843.8
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋

值给 $_，直到所有行都处理完毕。

2. 如果标量 $_ 中含有数字 6，则求表达式值，并用它代替替换操作的替换端内容。即 6 乘以 7.3（e 修饰符）。每次碰到 6 时，就用上述乘积（43.6）代替原数字 6（g 修饰符）。
3. 打印每一行内容。最后一行中含有两个 6，它们都会被替换为 43.8。

示例 8.31

```
(The Script)
    # The e modifier
1   $_=5;
2   s/5/6 * 4 - 22/e;
3   print "The result is: $_\n";

4   $_=1055;
5   s/5/3*2/eg;
6   print "The result is: $_\n";

(Output)
3   The result is: 2
6   The result is: 1066
```

解释

1. 将标量 $_ 赋值为 5。
2. 运算符 s 负责在 $_ 中搜索正则表达式 5。修饰符 e 表明要对数字表达式求值，并用算术运算 6*4-22（等于 2）的结果替换原字符串。
3. 打印求值结果。
4. 将标量 $_ 赋值为 1055。
5. 运算符 s 负责在 $_ 中搜索正则表达式 5。修饰符 e 表明要对数字表达式求值，并用乘积 3*2 的值替换它。也就是说，每当碰到 5 时，就将它替换为 6。由于替换操作是全局的，因此所有的 5 都会替换成 6。
6. 打印求值结果。

示例 8.32

```
(The Script)
1   $_ = "knock at heaven's door.\n";
2   s/knock/"knock, " x 2 . "knocking"/ei;
3   print "He's $_;

(Output)
He's knock, knock, knocking at heaven's door.
```

解释

1. 标量 $_ 中存有字符串 Knock at heaven's door.\n。
2. 运算符 s 负责在 $_ 中搜索正则表达式 knock。修饰符 e 表明要对字符串表达式求值，将其替换为 knock x 2（即重复 2 次），然后与字符串 knocking 相连（点运算符）。整个过程忽略大小写。
3. 打印得到的字符串。

示例 8.33

```
(The Script)
```

```
     # Saving in the $& special scalar
1    $_=5000;
2    s/$_/$& * 2/e;
3    print "The new value is $_\.n";

4    $_="knock at heaven's door.\n";
5    s/knock/"$&," x 2 . "$&ing"/ei;
6    print "He's $_";
(Output)
3    The new value is 10000.
6    He's knock,knock,knocking at heaven's door.
```

解释

1. 将标量 $_ 赋值为 5000。
2. 搜索字符串 5000 保存在变量 $& 中。求替换端表达式的值，即 $& 的值乘以 2。然后用新的值替换原有值，并给 $_ 赋新值。
3. 打印结果值。
4. 将标量 $_ 赋值为字符串 Knock at heaven's door.\n。
5. 如果找到了搜索字符串（knock）的话，将其存入标量 $& 中。求替换端表达式的值，即重复两次 $& 的值（knock）并与 $&ing（knocking）相连。然后用新的值替换原有值，给 $_ 赋新值并打印。

8.3.3 模式绑定运算符

模式绑定（pattern binding）运算符能够把匹配的模式、替换或转换（详见附录 A 中的 tr）与另一个标量表达式绑定到一起。在前面的示例中，基于默认模式空间 $_ 标量上的模式搜索过程是隐式（或显式）完成的。即，循环处理的每一行文件内容都必须保存在变量 $_ 中。在前面所述示例中，都是为 $_ 赋一个值，并将其用作替换时的搜索字符串。但是，如果程序需要把这个值保存到 $_ 之外的某个变量，那又该怎么办呢？

即不使用

```
$_ = 5000;
```

而应当使用

```
$salary = 5000;
```

如果要在 $salary 上执行模式匹配或替换操作，就不能使用

```
print if /5/; 或者 s/5/6;
```

而是应当使用

```
print if $salary = ~/5/; 或者 $salary = ~s/5/6;
```

因此，如果不把字符串存入 $_ 变量，同时又想对它执行模式匹配或替换操作的话，就应当使用模式绑定运算符 =~ 或 !=。在 tr 函数中，它们还可用于字符串翻译过程。

表 8-3 列出了所有模式匹配运算符。

格式

```
Variable =~ /Expression/
Variable !~ /Expression/
Variable =~ s/old/new/
```

表 8-3　模式匹配运算符

示例	涵　义
$name =~/John/	如果 $name 含有模式则为真。如果是真，返回 1；否则返回空值
$name !~/John/	如果 $name 不含有模式，则为真
$name =~s/John/Sam/	将匹配 John 的第一个值替换为 Sam
$name =~s/John/Sam/g	将匹配 John 的所有具体值替换为 Sam
$name =~tr/a-z/A-Z/	将所有小写字符翻译为大写字母
$name =~/$pal/	在搜索字符串时使用变量

示例 8.34

```
(The Script)
    # Using the $_ scalar explicitly
1   while($_=<DATA>){
2       print $_ if $_ =~ /Igor/;  # $_ holds the current input line
3   #   print if /Igor/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Igor Chevsky
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 如果有正则表达式 /Igor/ 在变量 $_ 中匹配，则通过 print 函数打印 $_ 的值。只有在显式地使用 $_ 作为操作数时才需要 =~。
3. 如果省略了模式匹配运算符 =~，则默认为与 $_ 相匹配。如果没有给 print 函数提供参数的话，默认也会打印 $_ 的值。

示例 8.35

```
(The Script)
    #!/usr/bin/perl
1   $name="Tommy Tuttle";
2   print "Hello Tommy\n"  if $name =~ /Tom/;
```

```
                                            # Prints Hello Tommy,if true
3   print "$name\n" if  $name !~ /Tom/;  # Prints nothing if false

4   $name =~ s/T/M/;                   # Substitute first T with an M
5   print "$name.\n";

6   $name="Tommy Tuttle";
7   print "$name\n" if $name =~ s/T/M/g;  # Substitute every T with M
8   print "What is Tommy's last name? ";
9   print "You got it!\n" if <STDIN> =~ /Tuttle/;

(Output)
2   Hello Tommy
5   Mommy Tuttle.
7   Mommy Muttle
8   What is Tommy's last name? Tuttle
9   You got it!
```

解释

1. 将标量 $name 赋值为字符串 Tommy Tuttle。
2. 若 $name 中含有模式 Tom，则打印字符串 $name。在成功匹配时，返回值为 1。
3. 若 $name 中不含模式 Tom，就不打印字符串 $name。不成功匹配时的返回值为空值。
4. 将 $name 中第一个字母 T 替换为字母 M。
5. 打印 $name，显示替换的效果。
6. 将标量 $name 赋值为字符串 Tommy Tuttle。
7. 将 $name 中的所有字母 T 替换为字母 M。位于替换表达式末尾的修饰符 g 负责产生跨行的全局替换效果。
8. 要求用户提供输入。
9. 针对正则表达式 Tuttle，匹配用户输入（<STDIN>）。如果发现匹配项，则执行 print 函数。

示例 8.36

```
(The Script)
1   $salary=50000;
2   $salary =~ s/$salary/$& * 1.1/e;
3   print "\$& is $&\n";
4   print "The salary is now \$$salary.\n";

(Output)
3   $& is 50000
4   The salary is now $55000.
```

解释

1. 将标量 $salary 赋值为 5000。
2. 在 $salary 上执行替换操作，并求替换端表达式的值。特殊标量 $& 中含有搜索端找到的值。若要在匹配完成后改变 $salary 的值，则应当使用模式匹配运算符 =~，并将替换结果与标量 $salary 结合。
3. 标量 $& 中含有在替换搜索端找到的内容。
4. 把标量 $salary 的值增加 10%。

示例 8.37

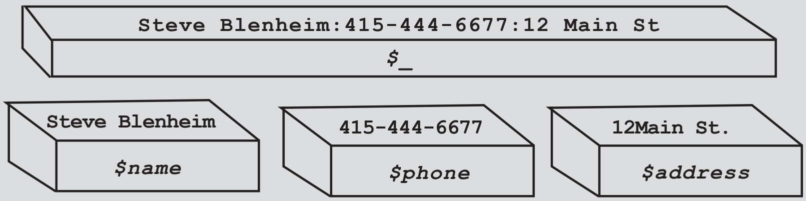
```
(The Script)
    # Using split and pattern matching
1   while(<DATA>){
2       @line = split(":", $_);
3       print $line[0],"\n" if $line[1] =~ /408-/
                                      # Using the pattern matching operator
    }
4   _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.

(Output)
Igor Chevsky
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 基于冒号拆分来自文件的每一行内容，并将返回值存入数组 @line 中。
3. 针对数组元素 $line[1] 匹配模式 /408-/。如果该模式在 $line[1] 中匹配，就打印 $line[0] 的内容。由于保存在 $line[1] 中的电话号码能与区号 408 成功匹配，因此这里将打印 $line[0] 的值，即 Igor 的名字。
4. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.38




```
(The Script)
    # Using split, an anonymous list, and pattern matching
    while(<DATA>){
        ($name, $phone, $address) = split(":", $_);
            print $name if $phone =~ /408-/    # Using the pattern
                                               # matching operator
    }
    _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.
```

```
(Output)
Igor Chevsky
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 基于冒号拆分来自文件的每一行内容，并将返回值存入由三个标量 $name、$phone 和 $address 组成的匿名列表里。这里的匿名列表比前面示例中的数组更易于理解和操作。在使用数组时，读者必须保证下标的正确性；而使用匿名列表时，该用哪个标量则是显而易见的。
3. 对变量 $phone 匹配模式 /408-/。如果该模式在 $phone 中匹配，就打印 $name 的值。由于本例中其电话号码与区号 408 相匹配，所以会打印 Igor。
4. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

示例 8.39

```
(The Script)
1   while($inputline=<DATA>){
2       ($name, $phone, $address) = split(":", $inputline);
3       print $name if $phone =~ /^408-/;  # Using the pattern
                                           # matching operator
4       print $inputline if $name =~ /^Karen/;
5       print if /^Norma/;
    }

6   _ _DATA_ _
    Steve Blenheim:415-444-6677:12 Main St.
    Betty Boop:303-223-1234:234 Ethan Ln.
    Igor Chevsky:408-567-4444:3456 Mary Way
    Norma Cord:555-234-5764:18880 Fiftieth St.
    Jon DeLoach:201-444-6556:54 Penny Ln.
    Karen Evich:306-333-7654:123 4th Ave.

(Output)
3   Igor Chevsky
4   Karen Evich:306-333-7654:123 4th Ave.
5   < No output >
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 基于冒号拆分来自 $inputfile 文件的每一行内容，并将返回值存入由三个标量 $name、$phone 和 $address 组成的匿名列表里。
3. 对变量 $phone 匹配模式 /408-/。如果该模式在 $phone 中匹配，就打印 $name 的值。由于本例中其电话号码与区号 408 相匹配，所以会打印 Igor。
4. 将每一行内容保存到 $inputline 变量中，逐行进行处理，直到文件末尾。如果其内容以正则表达式 Karen 开头，则打印变量 $inputline 的值。
5. 由于这里不再使用默认的行占位符 $_，因此无需为它赋值，也不用对它执行匹配或输出操

作。现在使用的是用户定义的 $inputline 变量，由它负责存储并匹配模式。
6. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。

8.4 读者应当学到的知识

1. 正则表达式的含义是什么？
2. 如何使用 if 和 unless 修饰符？
3. 如何才能在执行模式搜索时把正斜杠定界符改为其他字符？
4. s 运算符能完成哪些操作？
5. 全局搜索的含义是什么？
6. 何时需要使用模式绑定运算符 = ~ 和 !~ ？
7. 默认模式空间中保存了什么内容？
8. 文件句柄 _DATA_ 的用途是什么？
9. 修饰符 i、e、g 分别表示什么意思？

8.5 下章简介

在下一章里，读者将学习 Perl 提供的丰富的正则表达式元字符，并领教模式匹配的巨大威力。读者将学习如何标定（anchor）模式，如何搜索其他可用模式、空白字符、字符组、重复模式等。读者还将学习有关贪婪元字符的知识，以及如何控制它们。此外，还将介绍如何通过模式捕获与模式分组实现向前或向后的查询操作。在学完下一章内容后，读者将能基于指定场景，用正则表达式搜索目标数据，进而实现数据的验证，或者修改找到的文本内容。

练习 8

（读者可在华章网站 (www.hzbook.com) 上找到如下 sample.file 文件）

```
Tommy Savage:408-724-0140:1222 Oxbow Court, Sunnyvale,CA 94087:5/19/66:34200
Lesle Kerstin:408-456-1234:4 Harvard Square, Boston, MA 02133:4/22/62:52600
JonDeLoach:408-253-3122:123 Park St., San Jose, CA 94086:7/25/53:85100
Ephram Hardy:293-259-5395:235 Carlton Lane, Joliet, IL 73858:8/12/20:56700
Betty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA 91464:6/23/23:14500
William Kopf:846-836-2837:6937 Ware Road, Milton, PA 93756:9/21/46:43500
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI 23874:3/28/45:245700
James Ikeda:834-938-8376:23445 Aster Ave., Allentown, NJ 83745:12/1/38:45000
Lori Gortz:327-832-5728:3465 Mirlo Street, Peabody, MA 34756:10/2/65:35200
Barbara Kerz:385-573-8326:832 Ponce Drive, Gary, IN 83756:12/15/46:268500
```

1. 打印所有含有模式 Street 的行。
2. 打印名字与 B 或 b 相匹配的行。
3. 打印与 Ker 匹配的行。
4. 打印区号为 408 的电话号码。

5. 打印 Gortz 的名字和地址。
6. 以大写形式打印 Ephram 的名字。
7. 打印不含 4 的行。
8. 将 William 的名字改为 Siegfield。
9. 打印 Tommy Savage 的生日。
10. 打印收入超过 $40,000 的名字。
11. 打印在 6 月出生的人的名字。
12. 打印 Massachusetts 的邮件编码。

第 9 章　正则表达式元字符

9.1　正则表达式元字符

正则表达式元字符（metacharacter）是不代表自身原有含义的字符。它们拥有以某种方式控制搜索模式的特殊能力（例如只在行首或行尾搜索模式，或只在以大写或小写字母开头的行上搜索模式）。如果在它们前面加上反斜杠（\），这些元字符就会失去其特殊含义。例如，元字符点号（.）代表任何单个字符，但如果在前面加上反斜杠，它就会退化为一个普通的点号或句号。

如果在元字符前面出现了反斜杠，这些反斜杠就会关闭元字符的特殊含义；但如果在正则表达式中的其他数字或字母之前出现反斜杠的话，这些反斜杠则会拥有其他的含义。Perl 为一些元字符提供了简化形式，又称元符号（metasymbol），它们专门用于表示字符。例如，[0-9] 表示范围从 0 到 9 的数字，\d 也能表示相同的含义。只不过 [0-9] 用到了方括号元字符，而 \d 则使用了元符号。

示例 9.1

```
/^a...c/
```

解释

这个正则表达式含有元字符（详见表 9.1）。其中第一个元字符是脱字符（caret，即 ^）。脱字符只有在位于行首时才能用于匹配字符串。点号（.）用于匹配任意单个字符，包括空白字符。该表达式中含有三个点号，可代表任意三个字符。如需查询实际的点号或其他不代表其自身值的字符，应当在这些字符前加上反斜杠，以免混淆。

在示例 9.1 中，正则表达式可以读成：在行首搜索字母 a，其后紧跟任意三个字符，最后则是一个 c。譬如，如果在行首找到模式的话，它将与 abbbc、a123c、a c 或 aAx3c 匹配。

表 9-1　元字符

元　字　符	匹　配　项
字符类：单字符与数字	
.	匹配除换行符之外的任意字符
[a-z0-9]	匹配集合中任意单字符

（续）

<table>
<tr><th>元　字　符</th><th>匹　配　项</th></tr>
<tr><td>[^a-z0-9]</td><td>匹配不在集合中的任意单字符</td></tr>
<tr><td>\d</td><td>匹配单个数字</td></tr>
<tr><td>\D</td><td>匹配非数字字符，等效于 [^0-9]</td></tr>
<tr><td>\w</td><td>匹配数字型的（字）字符</td></tr>
<tr><td>\W</td><td>匹配非数字型的（非字）字符</td></tr>
<tr><td>字符类：空白字符</td><td></td></tr>
<tr><td>\s</td><td>匹配空白字符，如空格、制表符和换行符</td></tr>
<tr><td>\S</td><td>匹配非空白字符</td></tr>
<tr><td>\n</td><td>匹配换行符</td></tr>
<tr><td>\r</td><td>匹配回车符</td></tr>
<tr><td>\t</td><td>匹配制表符</td></tr>
<tr><td>\f</td><td>匹配进纸符</td></tr>
<tr><td>\b</td><td>匹配退格符</td></tr>
<tr><td>\0</td><td>匹配空值字符</td></tr>
<tr><td>字符类：锚定字符</td><td></td></tr>
<tr><td>\b</td><td>匹配字边界（不在 [] 中时）</td></tr>
<tr><td>\B</td><td>匹配非字边界</td></tr>
<tr><td>^</td><td>匹配行首</td></tr>
<tr><td>$</td><td>匹配行尾</td></tr>
<tr><td>\A</td><td>匹配字符串开头</td></tr>
<tr><td>\Z</td><td>匹配字符串或行的末尾</td></tr>
<tr><td>\z</td><td>只匹配字符串末尾</td></tr>
<tr><td>\G</td><td>匹配前一次 m//g 离开之处</td></tr>
<tr><td>字符类：重复字符</td><td></td></tr>
<tr><td>x?</td><td>匹配 0 或 1 个 x</td></tr>
<tr><td>x*</td><td>匹配 0 或多个 x</td></tr>
<tr><td>x+</td><td>匹配 1 或多个 x</td></tr>
<tr><td>(xyz)+</td><td>匹配 1 或多个模式 xyz</td></tr>
<tr><td>x(m,n)</td><td>匹配 m 到 n 个 x 组成的值</td></tr>
<tr><td>字符类：替换字符</td><td></td></tr>
<tr><td>was|were|will</td><td>匹配 was、were、will 之一</td></tr>
</table>

（续）

元　字　符	匹　配　项
字符类：记忆字符	
(stirng)	用于反向引用（详见示例 9.38 和 9.39）
\1 或 $1	匹配第一组括号 [a]
\2 或 $2	匹配第二组括号
\3 或 $3	匹配第三组括号
字符类：其他字符	
\12	匹配八进制数，直到 \377
\x811	匹配十六进制数值
\cX	匹配控制字符。譬如 \cC 指的是 <Ctrl>-C；\cV 指的是 <Ctrl>-V
\e	匹配 ASCII 编码中的 ESC 符（取消），而非反斜杠
\E	标识使用 \U、\L 或 \Q 的大小写更改操作的结束位置
\l	只小写下一个字符
\L	小写字符，直到字符串末尾或碰到 \E
\N	匹配已命名的字符，如 \N{greek:Beta}
\p{PROPERTY}	匹配拥有已命名属性的任意字符，譬如 \p{IsAlpha}/
\p{PROPERTY}	匹配不带已命名属性的任意字符
\Q	引用 \E 之前的元字符
\u	只大写下一个字符
\U	大写字符，直到字符串末尾或碰到 \E
\x{NUMBER}	匹配以十六进制形式给出的 Unicode 编码 NUMBER
\X	匹配 Unicode 编码“组合字符序列”字符串
\[	匹配元字符
\\	匹配反斜杠

a. \1 与 $1 称为反向引用（backreference）。其区别在于，\1 反向引用只在本模式中有效；而 $1 在代码块中或下一次成功匹配之前仍然有效。

9.1.1　表示单个字符的元字符

如需在正则表达式中查找某个特定字符，则可使用点号（.）来代表单个字符，或者借助字符类（character class）在一组字符中匹配单个字符。除了点号和字符类外，Perl 还提供了一些带有反斜杠的符号（又称作元符号，metasymbol），它们也能代表单个字符（详见表 9-2）。

表 9-2　元字符

元字符	匹配项
.	匹配除换行符之外的任意字符
[a-z0-9_]	匹配集合中任意单字符
[^a-z0-9_]	匹配不在集合中的任意单字符
\d	匹配单个数字
\D	匹配非数字字符，等效于 [^0-9]
\w	匹配数字型的（字）字符，等效于 [a-z0-9]
\W	匹配非数字型的（非字）字符，等效于 [^a-z0-9]

点号元字符。 点号（.）元字符能够匹配除换行符之外的任意单个字符。譬如，如果字符串中含有 a，其后是任意一个字符（\n 除外），最后是一个 b 的话，它就能与正则表达式 /a.b/ 相匹配；而表达式 /.../ 将会匹配含有至少三个字符的任意字符串。

示例 9.2

```
(The Script)
    # The dot metacharacter
1   while(<DATA>){
2       print "Found Norma!\n" if /N..ma/;

    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Found Norma!
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有在 $_ 中找到指定模式时，才打印字符串 Found Norma ！ \n。目标模式以大写字母 N 开头，后面是两个任意字符，最后则是 m 与 a。借助该模式便可找到 Norma、No man 和 Normandy 等结果。

s 修饰符：点号元字符与换行符。一般而言，点号元字符不会与换行符 \n 相匹配，因为它只能匹配字符串中位于换行符之前的字符。s 修饰符（modifier）能把嵌有换行符的行当作独立的一行来处理，而不是当作一组多个行；并允许点号元字符像处理其他字符一样处理换行符。s 修饰符可用在 m（match，匹配）运算符和 s（substitution，替换）运算符上。

示例 9.3

```
(The Script)
    # The s modifier and the newline
1   $_="Sing a song of sixpence\nA pocket full of rye.\n";
2   print $& if /pence./s;
3   print $& if /rye\../s;
4   print if s/sixpence.A/twopence, a/s;

(Output)
2   pence
3   rye.
4   Sing a song of twopence, a pocket full of rye.
```

解释

1. 为标量 $_ 赋值。它含有两个换行符。
2. 正则表达式 /pence./ 中含有点号元字符。这里的点号元字符不能与换行符匹配，除非使用 s 修饰符。特殊变量 $& 中含有在上次成功的模式搜索过程中找到的值，即 pence\n。
3. 正则表达式 /rye\.../ 中含有一个实际的点号（实际点号由反斜杠负责标识），其后是能与换行符匹配的点号元字符。这是 s 修饰符在起作用。特殊变量 $& 中含有在上次成功的模式搜索过程中找到的值，即 rye.\n。
4. s 修饰符使得点号元字符能够匹配于搜索字符串中的换行符，并将换行符替换为空格。

字符类。字符类代表了字符集合中的一个字符。例如，[abc] 匹配 a、b 或 c；[a-z] 匹配从 a 到 z 的字符集中的某一字符；而 [0-9] 则匹配从 0 到 9 数字中的一个字符。如果在字符前面带有前导脱字符（^），则表示任何不在该集合中的字符。例如，[^a-zA-Z] 匹配不在 a 到 z 或 A 到 Z 范围内的单个字符；[^0-9] 则匹配不在 0 到 9 范围内的一个数字字符。如要匹配从 10 到 13 之间的数字字符[⊖]，则应使用 1[0-3]，而不是 [10-13]。

Perl 提供了一组附加符号，即元符号（metasymbol），用于表示字符类。符号 \d 和 \D 分别代表单个数字和一个非数字的字符；它们分别等效于 [0-9] 和 [^0-9]。同样，\w 和 \W 分别代表单字字符和非单字字符，它们与 [A-Za-z_0-9] 和 [^A-Za-z_0-9] 效果相同。

示例 9.4

```
(From a Script)
1   while(<DATA>){
2       print if /[A-Z][a-z]eve/;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Steve Blenheim 101
```

⊖ 请读者不要将方括号中的脱字符与用于行锚定（line anchor）起始位置的脱字符相混淆。详见表 9.7。

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配特定模式时才打印 $_ 中的行内容，该模式以一个大写字母 [A-Z] 开头，随后是一个小写字母 [a-z]，最后则是字符串 eve。

示例 9.5

```
(The Script)
    # The bracketed character class
    while(<DATA>){
        print if /[A-Za-z0-9_]/;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Steve Blenheim 101
Betty Boop 201
Igor Chevsky 301
Norma Cord 401
Jonathan DeLoach 501
Karen Evich 601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配于字符类 [A-Za-z0-9] 中的字母或数字字符时，才打印 $_ 中的行内容。

示例 9.6

```
(The Script)
    # The bracket metacharacters and negation
    while(<DATA>){
        print if / [^123]0/
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Norma Cord 401
```

```
Jonathan DeLoach 501
Karen Evich 601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配指定模式时才打印 $_ 中的行内容，该模式以一个空格开头，随后是一个不在 1 到 3 之间的数字（除 1、2、3 之外的数字），最后是一个 0。

示例 9.7

```
(The Script)
    # The metasymbol, \d
1   while(<DATA>){
2       print if /6\d\d/
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601
(Output)
Karen Evich 601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 当 $_ 匹配于数字 6 后面跟有两个数字字符的模式时，打印 $_ 中保存的行内容。元字符 \d 代表了字符类 [0-9]。

示例 9.8

```
(The Script)
    # Metacharacters and metasymbols
1   while(<DATA>){
2       print if /[ABC]\D/
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Steve Blenheim 101
Betty Boop 201
Igor Chevsky 301
```

```
Norma Cord 401
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配于指定模式时才打印 $_ 中的行内容，该模式是一个大写的 A、B 或 C，即 [ABC]。元字符 \D 等效于字符类 [^0-9]；即不在 0 到 9 范围内的数字字符。

示例 9.9

```
(The Script)
    # The word metasymbols
    while(<DATA>){
        print if / \w\w\w\w \d/
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Betty Boop 201
Norma Cord 401
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配特定模式时才打印 $_ 中的行内容，该模式以一个空格开头，其后是四个字母或数字字符 \w，再后是一个空格与一个数字字符。元信息 \w 等效于字符类 [A-Za-z0-9]。

示例 9.10

```
(The Script)
    # The word metasymbols
    while(<DATA>){
        print if /\W\w\w\w\w\W/
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601
(Output)
Betty Boop 201
Norma Cord 401
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 只有当 $_ 匹配特定模式时才打印 $_ 中的行内容，该模式含有一个非字母数字型字符，随后是四个字母数字型字符 \w，最后则是另一个非字母数字型字符 \W。元符号 \W 代表字符类 [^A-Za-z0-9]。Boop 和 Cord 都是位于空白字符之间（非字母数字字符）的四个字符。

POSIX 字符类。Perl5.6 引入了 POSIX 字符类。POSIX（Portable Operating System Interface ⊖）是一种保障程序间跨平台可移植性的工业标准。为了支持可移植性，POSIX 认识到不同国家和地区在字符编码方式、货币表示符号以及日期和时间表达方式上都有所不同。要处理这些不同类型的字符，POSIX 为正则表达式提供了带方括号的字符类，如表 9-3 所示。

类 [:alnum:] 是表示 A-Za-z0-9 的另一种方式。若要使用该字符类，必须将它放入另外一组方括号中，以便识别为正则表达式。例如，A-Za-z0-9 本身并不是正则表达式字符类，但 [A-Za-z0-9] 是。同样，[:alnum:] 应当写作 [[:alnum:]]。使用前者 [A-Za-z0-9] 和后者 [[:alnum:]] 的区别在于，前者只受到 ASCII 字符编码的影响，而后者则允许在类中表示其他语言的相应字符。

如需在 POSIX 字符类中对字符取反，则可使用如下语法：

[^[:space:]] — 表示所有非空白的字符。

表 9-3　带方括号的字符类

方括号类	含　义
[:alnum:]	字母数字型字符
[:alpha:]	字母型字符
[:ascii:]	码值在 0 到 127 之间的字符
[:cntrl:]	控制字符
[:digit:]	数字字符，即 0-9 或者 \d
[:graph:]	除字母数字或标点符号之外的非空字符（即非空格、控制字符等）
[:lower:]	小写字母
[:print:]	与 [:graph:] 相似，但包含空格字符
[:punct:]	标点符号字符
[:space:]	所有空白字符（换行符、空格符、制表符等）
[:upper:]	大写字母
[:word:]	任意字母数字或下划线字符 [a]
[:xdigit:]	十六进制数字中允许的阿拉伯数字（0-9a-fA-F）

a. 这是 Perl 提供的扩展，等效于 \w。

⊖ POSIX 是 IEEE 的一个注册商标。详见 http://www.opengroup.org/austin/papers/backgrounder.html。

示例 9.11

```
(In Script)
    # The POSIX character classes
1   require 5.6.0;
2   while(<DATA>){
3       print if /[[:upper:]][[:alpha:]]+ [[:upper:]][[:lower:]]+/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Betty Boop
    Karen Evich
(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
Betty Boop
Karen Evich
```

解释

1. 如需使用 POSIX 字符类，Perl 的版本必需是 5.6.0（或以上）。
2. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
3. 正则表达式含有 POSIX 字符类。只有当 $_ 匹配指定的模式时才打印 $_ 中的行内容，该模式必须含有任意一个大写字母 [[:upper:]]，随后是一到多个（+）字母字符 [[:alpha:]]，最后则是一个大写字母与一到多个小写字母。（加号+是表示一到多个字符的正则表达式元字符，本书将在“重复模式匹配元字符”一节中予以讨论。）

9.1.2 空白元字符

空白字符（whitespace character）包括空格符、制表符、回车符、换行符或进纸符。用户可通过按下 <Tab> 键、空格键或者 <Enter> 键得到实际的空白字符。

表 9-4 空白元字符

元字符	匹 配 项
\s	匹配空白字符，如空格、制表符和换行符
\S	匹配非空白字符
\n	匹配换行符或行末符（UNIX：012；Mac OS：015）
\r	匹配回车符
\t	匹配制表符
\f	匹配进纸符

示例 9.12

```
(The Script)
    # The \s metasymbol and whitespace
    while(<DATA>){
        print if s/\s/*/g; # Substitute all spaces with stars
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Steve*Blenheim*101*Betty*Boop*201*Igor*Chevsky*301*Norma*
*Cord*401*Jonathan*DeLoach*501*Karen*Evich*601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 如果 $_ 的内容与含有空白字符（即空格、制表符或换行符）\s 的模式相匹配，则打印该行内容。最后将所有空白字符替换为星号（*）。

示例 9.13

```
(The Script)
    # The \S metasymbol and nonwhitespace
    while(<DATA>){
        print if s/\S/*/g;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601
(Output)
***** ******** ***
***** **** ***
**** ******* ***
***** **** ***
******** ******* ***
***** ****** ***
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 如果 $_ 的内容与含有非空白字符（即不是空格、制表符或换行符）\S 的模式相匹配，则打印该行内容。然后将所有空白字符替换为 *。大写的元字符等效于对相应的小写元字符取反，譬如 \d 表示数字，\D 表示非数字。

示例 9.14

```
(The Script)
    # Escape sequences, \n and \t
    while(<DATA>){
        print if s/\n/\t/;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Steve Blenheim 101   Betty Boop 201   Igor Chevsky 301
Norma Cord 401   Jon DeLoach 501   Karen Evich 601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有 \n 转义序列，代表单个换行符。表达式读作：以制表符（\t）替换每个换行符。

9.1.3 重复模式匹配元字符

在前面的示例中，元字符都是匹配于单个字符的。如果需要匹配多个字符，又该怎么办呢？譬如，需要找到所有含有特定名字的行，该名字必须以大写字母开头（表示为 A − Z），后面跟随一系列小写字母，并且名字长度各不相同。[a-z] 只能匹配单个小写字母。那该如何匹配一到多个小写字母，或零到多个字母呢？这时用户可使用所谓的限定符（quantifier）。如需匹配一到多个小写字母，正则表达式可以写成 /[a-z]+/，其中加号＋的意思是“一到多个前面出现的字符”，在这里就是一到多个字母。Perl 还提供了其他多种限定符，见表 9-5。

表 9-5 贪婪（greedy）元字符

元字符	匹配项
x?	匹配 0 至 1 个 x 组成的具体值
(xyz)?	匹配 0 至 1 个 xyz 模式组成的具体值
x*	匹配 0 至多个 x 组成的具体值
(xyz)*	匹配 0 至多个 xyz 模式组成的具体值
x+	匹配 1 至多个 x 组成的具体值
(xyz)+	匹配 1 至多个 xyz 模式组成的具体值
x{m,n}	匹配多于 m 个且少于 n 个的 x 所组成的具体值

贪婪因子。一般而言，限定符都是贪婪（greedy）的。也就是说，它们会从左到右搜索能够匹配的最长字符集，并获得最后一个满足条件的字符。例如，给出下面这个字符串：

```
$_="ab123456783445554437AB"
```

还有正则表达式：

```
s/ab[0-9]*/X/;
```

则搜索端的匹配结果为：

```
ab123456783445554437
```

所有这些内容都将被一个 X 替换。替换后 $_ 的值变为：

```
XAB
```

星号（*）是一个贪婪元字符，负责匹配 0 到多个前面出现的字符。在上述示例中，星号匹配的是 [0-9] 字符类。该匹配动作会从最左边开始搜索 ab 及其右侧的零到多个范围在 0 到 9 之间的数字。匹配动作一直持续到碰到最后一个连续数字 7 为止。最后，把模式 ab 以及其后所有数字一起替换为 X。

用户也可关闭贪婪性。这样就不再匹配最大数目的字符，而是与找到的最少数目的字符相匹配。这也可通过在贪婪元字符后面追加问号（?）的方式实现。详见示例 9.15。

示例 9.15

```
(The Script)
    # The zero or one quantifier
1   while(<DATA>){
2       print if / [0-9]\.?/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Steve Blenheim 1.10
Igor Chevsky 555.100
Norma Cord 4.01
Karen Evich 601
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有元字符 ?，代表零到一个前面出现的字符。该表达式读作：查找一个空格符，后面跟有一个位于 0 到 9 之间的数字，最后则是 0 个或 1 个实际的点号。

示例 9.16

```
(The Script)
    # The zero or more quantifier
    while(<DATA>){
        print if /\sB[a-z]*/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Steve Blenheim 1.10
Betty Boop .5
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有元字符 *，代表零到多个前面的字符。表达式读作：查找一个空格 \s，随后是一个大写字母 B，最后则是零到多个小写字母，即 [a-z]*。

示例 9.17

```
(The Script)
    # The dot metacharacter and the zero or more quantifier
     while(<DATA>){
        print if s/[A-Z].*y/Tom/;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
Tom Boop 201
Tom 301
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有元字符 *，代表零到多个前面的字符。本例中，前面的字符是点号，本身就代表任何字符。该表达式读作：查找一个大写字母 [A-Z]，随后是零到多个任意字符，最后则是小写字母 y。如果该行含有多个 y，则搜索结果将包含所有的 y，直到最后一个 y 为止。Betty 和 Ignor Chevsky 都能满足匹配条件。请注意，Ignor Chevsky 中的空格是因为匹配了点号才出现在结果中的。

示例 9.18

```
(The Script)
    # The one or more quantifier
1   while(<DATA>){
2       print if /5+/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Betty Boop .5
Igor Chevsky 555.100
Jonathan DeLoach .501
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有元字符 +，代表一到多个前面的字符。表达式读作：查找由一个到多个 5 组成的具体匹配值。

示例 9.19

```
(The Script)
    # The one or more quantifier
1   while(<DATA>){
2       print if s/\w+/X/g;
    }
    _ _DATA_ _
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jonathan DeLoach 501
    Karen Evich 601

(Output)
X X X
X X X
X X X
X X X
X X X
X X X
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。

2. 正则表达式中含有 \w，其后带有元字符 +，代表一个到多个字母或数字字符。例如，第一个字母数字字符集合是 Steve，所以会把 Steve 替换为 X。由于替换操作是全局的，因此下一组字母数字字符也将替换为 X。最后，把字母数字集合 101 也替换为 X。

示例 9.20

```
(The Script)
    # Repeating patterns
    while(<DATA>){
        print if /5{1,3}/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601
(Output)
Betty Boop .5
Igor Chevsky 555.100
Jonathan DeLoach .501
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有花括号（{}）元字符，代表前面字符重复的次数范围。表达式读作：在行中查找由不少于 1 个且不多于 3 个连续数字 5 组成的具体匹配值。

示例 9.21

```
(The Script)
    # Repeating patterns
    while(<DATA>){
        print if /5{3}/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Igor Chevsky 555.100
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。

2. 表达式读作：查找连续出现三个 5 的具体值。这并不是要求字符串中正好出现三个数字 5，而是意味着至少要有三个连续的 5。如果字符串含有 5555555，则该匹配仍然可以成功。如需查找正好出现三个数字 5 的情况，则应通过某种方式锚定（anchor）该模式，即使用 ^ 和 $ 锚定元字符；或者也可在三个连续数字 5 前后放置其他一些字符，譬如：/^5{3}$/ 或 /5{3}898/ 或 /95{3}\.56/。

示例 9.22

```
(The Script)
    # Repeating patterns
1   while(<DATA>){
2       print if /5{1,}/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Betty Boop .5
Igor Chevsky 555.100
Jonathan DeLoach .501
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 表达式读作：查找由一个或多个连续的数字 5 组成的具体值。

关闭贪婪性的元字符。如需关闭贪婪性，用户只需在贪婪限定符后面加上问号便可。这样就能让搜索操作在碰到第一个匹配时就告一段落，而不是直到最后一个匹配。详见表 9.6。

示例 9.23

```
(The Script)
    # Greedy and not greedy
1   $_="abcdefghijklmnopqrstuvwxyz";
2   s/[a-z]+/XXX/;
3   print $_, "\n";

4   $_="abcdefghijklmnopqrstuvwxyz";
5   s/[a-z]+?/XXX/;
6   print $_, "\n";

(Output)
3   XXX
6   XXXbcdefghijklmnopqrstuvwxyz
```

解释

1. 将标量 $_ 赋值为全部由小写字符组成的字符串。

2. 正则表达式读作：搜索一个或多个小写字母，然后将它们替换成 XXX。其中 + 元字符是贪婪的。它会获取与表达式匹配的所有字符，即：从字符串最左侧开始，获得能够碰到的所有小写字母，直到字符串的末尾。
3. 打印替换完毕后的 $_ 变量值。
4. 将标量 $_ 赋值为小写字符组成的字符串。
5. 正则表达式读作：搜索一个或多个小写字母，在碰到第一次匹配时便停止搜索，并将它替换成 XXX。元字符 + 后面附加的问号（?）负责关闭元字符的贪婪性，从而只搜索最少数目的匹配字符。
6. 打印替换完毕后的 $_ 变量值。

表 9-6　关闭贪婪性

元字符	匹配项
x??	匹配由 0 至 1 个 x 组成的具体值
(xyz)??	匹配由 0 至 1 个 xyz 模式组成的具体值
x*?	匹配由 0 至多个 x 组成的具体值
(xyz)*?	匹配由 0 至多个 xyz 模式组成的具体值
x+?	匹配由 1 至多个 x 组成的具体值
(xyz)+?	匹配由 1 至多个 xyz 模式组成的具体值
x{m,n}?	匹配由多于 m 个且少于 n 个的 x 所组成的具体值
x{m}?	匹配由至少 m 个 x 所组成的具体值
x{m,}?	匹配至少 m 次

示例 9.24

```
(The Script)
    # A greedy quantifier
1   $string="I got a cup of sugar and two cups of flour
             from the cupboard.";

2   $string =~ s/cup.*/tablespoon/;
3   print "$string\n";
    # Turning off greed
4   $string="I got a cup of sugar and two cups of flour
             from the cupboard.";
5   $string =~ s/cup.*?/tablespoon/;
6   print "$string\n";

(Output)
3   I got a tablespoon
6   I got a tablespoon of sugar and two cups of flour from the cupboard.
```

解释

1. 将标量 $string 赋值为含有三个连续 cup 模式的字符串。
2. s（替换）运算符将搜索模式 cup 及其后面的零到多个字符；即：匹配 cup 和到行末为止的所有字符，并将它们替换成 tablespoon。.* 限定符是贪婪的，因为它会匹配尽可能长的模式。

3. 输出贪婪替换后的结果。
4. 重新对 $string 进行赋值。
5. 这次搜索过程不是贪婪的。通过在 .* 限定符后追加一个问号，使得程序只把 cup 及其后面零到多个字符模式的最小匹配替换为 tablespoon。
6. 打印新字符串内容。

锚定元字符。锚定元字符是经常会用到的一类元字符，它能使模式匹配只发生在字符串的开头或末尾位置。这些元字符基于待匹配字符的左侧位置或右侧位置。锚（Anchor）在技术上又称作零宽度断言（assertion），因为它们的处理对象是位置，而不是字符串中实际的字符。例如，/^abc/ 的意思是在行首查找 abc，其中 ^ 代表了行首位置，而不是实际字符。

表 9-7　锚定（断言）

元字符	匹配项
^	匹配行或字符串的开头位置
$	匹配行或字符串的末尾位置
\A	只匹配字符串的开头位置
\Z	匹配字符串或行的末尾位置
\z	只匹配字符串的末尾位置
\G	匹配前面 m//g 模式的离开位置
\b	匹配一个字边界（当不位于 [] 中时）
\B	匹配一个非字边界

示例 9.25

```
(The Script)
    # Beginning of line anchor
1   while(<DATA>){
2       print if /^[JK]/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Jonathan DeLoach .501
Karen Evich 601.100
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。

2. 正则表达式中含有脱字符（^），只有当它是模式的第一个字符时，才代表行首锚定元字符。表达式读作：在行首位置搜索一个 J 或 K。\A 在本例中能产生与脱字符相同的字符集。表达式 /^[^JK]/ 读作：在行首位置搜索一个既不是 J 也不是 K 的字符。请记住，如果脱字符是在字符类中出现的话，就表示反转该字符类；如果是在起始限定符后直接出现的话，就表示锚定在行首位置。

示例 9.26

```
(The Script)
    # End of line anchor
1   while(<DATA>){
2       print if /10$/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.10
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Steve Blenheim 1.10
Igor Chevsky 555.10
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中含有元字符 $。当 $ 是模式的最后一个字符时，它就是指向行末位置的锚定元字符。表达式读作：查找后面跟有换行符的一个 1 和一个 0。

示例 9.27

```
(The Script)
    # Word anchors or boundaries
1   while(<DATA>){
2       print if /\bJon/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
Jonathan DeLoach .501
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容

赋值给 $_，直到所有行都处理完毕。

2. 正则表达式中含有 \b 元字符，它代表字边界（word boundary）。表达式读作：查找以 Jon 开头的字。

示例 9.28

```
(The Script)
    # Beginning and end of word anchors
1   while(<DATA>){
2       print if /\bJon\b/;
    }
    _ _DATA_ _
    Steve Blenheim 1.10
    Betty Boop .5
    Igor Chevsky 555.100
    Norma Cord 4.01
    Jonathan DeLoach .501
    Karen Evich 601

(Output)
<No output>
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式中也含有 \b 元字符，代表一个字边界。表达式读作：查找以模式 Jon 开头和结尾的字。结果这里什么都没有找到。

m 修饰符。m 修饰符用于控制锚定元字符 ^ 和 $ 的行为，它能把含有换行符的字符串当作多行内容来对待。如果使用元字符 ^ 锚定正则表达式，同时又在多行中的每一行开头找到了匹配模式的话，则也能匹配成功。同样，如果在多行中任意一行的末尾使用了元字符 $（或 \Z）来锚定正则表达式的话，只要找到该模式，就也能成功匹配。m 修饰符对于 \A 和 \z 是无效的。

示例 9.29

```
(The Script)
   # Anchors and the m modifier
1  $_="Today is history.\nTomorrow will never be here.\n";
2  print if /^Tomorrow/;     # Embedded newline

3  $_="Today is history.\nTomorrow will never be here.\n";
4  print if /\ATomorrow/;    # Embedded newline

5  $_="Today is history.\nTomorrow will never be here.\n";
6  print if /^Tomorrow/m;

7  $_="Today is history.\nTomorrow will never be here.\n";
8  print if /\ATomorrow/m;

9  $_="Today is history.\nTomorrow will never be here.\n";
10 print if /history\.$/m;
```

```
(Output)
6   Today is history.
    Tomorrow will never be here.
10  Today is history.
    Tomorrow will never be here.
```

解释

1. 将标量 $_ 赋值为含有换行符的字符串。
2. 元字符 ^ 会把搜索动作锚定在行的开头位置。由于本行不以 Tomorrow 开头，因此搜索失败，不返回任何内容。
3. 将标量 $_ 赋值为含有换行符的字符串。
4. 不论字符串的内容如何，\A 断言都只匹配该字符串的开头。由于本行不以 Tomorrow 开头，因此搜索失败，不返回任何内容。
5. 将标量 $_ 赋值为含有换行符的字符串。
6. m 修饰符把字符串当作多行内容对待，其中每一行都以一个换行符结尾。在本例中，元字符 ^ 能匹配其中任意一行的开头位置。本例在第二行的开头匹配到了模式 /^Tomorrow/。
7. 将标量 $_ 赋值为含有换行符的字符串。
8. \A 断言只匹配字符串的开头。字符串中嵌入的换行符和 m 修饰符在这里都是无效的。由于在字符串开头没有找到 Tomorrow，因此搜索失败，不返回任何内容。
9. 将标量 $_ 赋值为含有换行符的字符串。
10. 元字符 $ 会把搜索动作锚定在行的末尾位置。这里使用了 m 修饰符，把字符串当作多行来对待，其中每一行都以一个换行符结尾。本示例将在第一行末尾找到模式 /history\.$/。m 修饰符对 \Z 也是有效地，但它对 \z 无效。

交替（alternative）。交替特性允许正则表达式中出现能从多种可能中选中一种的模式匹配。譬如正则表达式 /John|Karen|Steve/ 就会与含有 John 或 Karen 或 Steve 的行相匹配。如果 John、Karen 和 Steve 位于不同的行，则所有相关的行都将获得匹配。每个交替正则表达式都由竖线符号（管道符号）隔开，并能匹配任意数目的字符。它不同于字符类，后者仅能匹配一个字符。例如，/a|b|c/ 等效于 [abc]，但 /ab|de/ 不能等效于 [abde]。因为模式 /ab|de/ 表示 ab 或者 de，而字符类 [abcd] 则表示集合中的单个字符，即 a、b、c 或者 d。

示例 9.30

```
(The Script)
    # Alternation: this, that, and the other thing
    while(<DATA>){
        print if /Steve|Betty|Jon/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jonathan DeLoach
    Karen Evich

(Output)
```

```
2    Steve Blenheim
     Betty Boop
     Jonathan DeLoach
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 出现在正则表达式中的管道符号（|）负责匹配一组交替（alternative）的模式。如果找到 Steve、Betty 或 John 当中任何一个模式的话，匹配就能成功。

分组（grouping）与分簇（clustering）。 如果正则表达式的模式位于括号内，便可以创建一个子模式。此后，表达式不再通过贪婪元字符去匹配零个或多个字符，而是去匹配前面定义的子模式。如果模式位于括号中，它也能控制交替性。Perl Wizard 把这种字符组合在一起的过程称作分簇（Clustering）。

示例 9.31

```
(The Script)
     # Clustering or grouping
1    $_=qq/The baby says, "Mama, Mama, I can say Papa!"\n/;
2    print if s/(ma|pa)+/goo/gi;

(Output)
The baby says, "goo, goo, I can say goo!"
```

解释

1. 将标量 $_ 赋值为带有双引号的字符串。
2. 正则表达式中含有位于括号内的模式，其后跟有元字符 +。括号与元字符 + 负责对其控制的字符进行分组。该表达式读作：查找连续一个或多个 ma 或 pa 模式所匹配的具体值，并将其替换为 goo。

示例 9.32

```
(The Script)
     # Clustering or grouping
1    while(<DATA>){
2        print if /\s(12){3}$/;  # Print lines matching exactly 3
                                 # consecutive occurrences of 12 at
                                 # the end of the line
     }
     _ _DATA_ _
     Steve Blenheim  121212
     Betty Boop      123
     Igor Chevsky    123444123
     Norma Cord      51235
     Jonathan DeLoach123456
     Karen Evich     121212456

(Output)
Steve Blenheim 121212
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 模式 12 是位于括号内的分组，它受限定符 {3} 控制。本例中，它将在行末（$）匹配恰好三个连续 12 组成的行。

示例 9.33

```
The Script)
    # Clustering or grouping
1   $_="Tom and Dan Savage and Ellie Main are cousins.\n";
2   print if s/Tom|Ellie Main/Archie/g;

3   $_="Tom and Dan Savage and Ellie Main are cousins.\n";
4   print if s/(Tom|Ellie) Main/Archie/g;

(Output)
2   Archie and Dan Savage and Archie are cousins.
4   Tom and Dan Savage and Archie are cousins.
```

解释

1. 将标量 $_ 赋值为一个字符串。
2. 如果在 $_ 中匹配了模式 Tom 或模式 Ellie Main，则将二者都替换成 Archie。
3. 将标量 $_ 赋值为一个字符串。
4. 本行把 Tom 和 Ellie Main 放在括号中间，使得交替模式变为 Tom Main 或 Ellie Main。由于标量 $_ 中只有 Ellie Main 能够成功匹配，所以将其替换为 Archie。

示例 9.34

```
(The Script)
    # Clustering and anchors
1   while(<DATA>){
2       # print if /^Steve|Boop/;
3         print if /^(Steve|Boop)/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jonathan DeLoach
    Karen Evich

(Output)
Steve Blenheim
```

解释

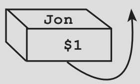
1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。

2. 该行已经注释掉了。打印任何以 Steve 开头的行，以及任何含有 Boop 的行。行首的锚定元字符和脱字符只对 Steve 有效。
3. 如果该行是以 Steve 或 Boop 开头的，则打印其内容。括号使得两个模式成为一个分组，因此行首的锚定元字符和脱字符对 Steve 与 Boop 都有效。也可将其写作 /(^Steve|^boop)/。

记忆或捕获。如果正则表达式位于括号中，则表示创建子模式。子模式保存在具有特殊编号的标量型变量中，首先是 $1，然后是 $2，依此类推。用户可在程序中使用这些变量，直到其他成功的匹配将它覆盖为止。如果需要控制贪婪元字符或前面示例所示交替模式的话，其子模式也将会随之保存，这也算是一种副作用吧[⊖]。

示例 9.35

```
(The Script)
   # Remembering subpatterns
   while(<DATA>){
       s/([Jj]on)/$1athan/;  # Substitute Jon or jon with
                             # Jonathan or jonathan
```




```
      print;
   }
   _ _DATA_ _
   Steve Blenheim
   Betty Boop
   Igor Chevsky
   Norma Cord
   Jon DeLoach
   Karen Evich

(Output)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jonathan Deloach
Karen Evich
```

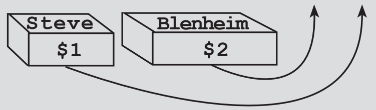
解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式含有一个位于括号内的模式 Jon。通过捕获该模式并将其保存到特殊变量 $1 中，便可完成该模式的记忆过程。如果括号内还有另一个模式的话，则将其保存到变量 $2 中，依此类推。替换端用到了这些特殊编号 $1、$2 和 $3 等。该表达式读作：查找 Jon 或 jon，并分别替换成 Jonathan 或 jonathan。在下一次成功匹配时，程序将清除这些特殊编号变量的内容。

⊖ 这里亦可阻止子模式的保存。

示例 9.36

```
(The Script)
   # Remembering multiple subpatterns
1  while(<DATA>){
2      print if s/(Steve)  (Blenheim)/$2, $1/
```




```
   }
   _ _DATA_ _
   Steve Blenheim
   Betty Boop
   Igor Chevsky
   Norma Cord
   Jonathan DeLoach
   Karen Evich

(Output)
Blenheim,Steve
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式含有两个位于括号中的模式。本行将捕获第一个模式，并将其保存到特殊标量 $1 中，然后捕获第二个模式并保存到特殊标量 $2 里。在替换端，由于首先引用的是 $2，因此程序将先打印 Blenheim，然后是一个逗号，最后则是 $1 的值，即 Steve（其作用是反转 Steve 与 Blenheim）。

示例 9.37

```
(The Script)
    # Reversing subpatterns
1   while(<DATA>){
2       s/([A-Z][a-z]+)\s([A-Z][a-z]+)/$2, $1/;
                                             # Reverse first and last names
3       print;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Blenheim, Steve
Boop, Betty
Chevsky ,Igor
Cord, Norma
De, JonLoach    # Whoops!
Evich, Karen
```

解释

该正则表达式也有两个位于括号内的模式。本例借助元字符来完成模式匹配过程。其第一个模式读作：查找一个大写字母后面跟随一个或多个小写字母的情况。其记忆的模式后面有一个空格。第二个模式读作：查找一个大写字母后面跟随一个或多个小写字母的情况。上述模式分别保存在特殊标量 $1 和 $2 中，然后在替换端予以反转。请注意其最后一个名字 DeLoach 会造成问题。这是因为 DeLoach 在第一个大写字母后面既有小写字母又有大写字母。如要支持这种情况，则应当把模式写作：s/([A-Z][a-z]+)\s([A-Z][A-Za-z]+)/$2,$1/。

示例 9.38

```
(The Script)
    # Metasymbols and subpatterns
1   while(<DATA>){
2       s/(\w+)\s(\w+)/$2, $1/; # Reverse first and last names
3       print;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Betty Boop

(Output)
Blenheim, Steve
Boop, Betty
Chevsky, Igor
Cord, Norma
DeLoach, Jon
Boop, Betty
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值 给 $_，直到所有行都处理完毕。
2. 正则表达式含有两个位于括号中的模式。其中 \w+ 代表一个或多个字。正则表达式由两个带括号的子模式（又称作反向引用）构成，并由一个空格符（\s）隔开。两个子模式分别保存在 $1 和 $2 中。在替换端通过 $1 与 $2 实现了第一个名字与最后一个名字的反转处理。

示例 9.39

```
(The Script)
    # Backreferencing
1   while(<DATA>){
2       ($first, $last)=/(\w+) (\w+)/;   # Could be: (\S+) (\S+)/
3       print "$last, $first\n";
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
```

```
    Norma Cord
    Jon DeLoach
    Betty Boop
(Output)
Blenheim, Steve
Boop, Betty
Chevsky, Igor
Cord, Norma
DeLoach, Jon
Boop, Betty
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式含有两个位于括号内的模式。其中 \w+ 代表一个或多个字。正则表达式由两个带括号的子模式（又称反向引用）构成。其返回值是一个由所有反向引用所组成的数组。分别将每个字赋值给变量 $first 与 $last。

示例 9.40

```
(The Script)
    # The greedy quantifier
1   $string="ABCdefghiCxyzwerC YOU!";
2   $string=~s/.*C/HEY/;
3   print "$string", "\n";

(Output)
HEY YOU!
```

解释

1. 将标量型变量 $string 赋值为含有很多模式 C 的字符串。
2. 替换操作的搜索端读作：查找以 C 结尾的字符数最大的任何模式。该搜索是贪婪的，它将从左到右搜索整个字符串，直到最后一个 C 为止。然后，把在 $string 中找到的内容替换为字符串 HEY。
3. 打印替换后的字符串，说明替换效果。

示例 9.41

```
(The Script)
    # Backreferencing and greedy quantifiers
1   $string="ABCdefghiCxyzwerC YOU!";
2   $string=~s/(.*C)(.*)/HEY/; # Substitute the whole string with HEY
3   print $1, "\n";
4   print $2, "\n";
5   print "$string\n";

(Output)
3   ABCdefghiCxyzwerC
4   YOU!
5   HEY
```

解释

1. 将标量 $string 赋值为字符串。
2. 正则表达式 /*.C/ 位于括号内。将找到的模式存储在特殊变量 $1 中，然后把剩余内容保存到 $2 内。
3. 把可能出现的最长模式保存在 $1 中，并打印该变量值。
4. 把剩余字符串保存在 $2 内，并打印该变量的值。
5. 完成替换操作，把整个字符串替换成 HEY。

示例 9.42

```
(The Script)
     # Backreferencing and greed
1    $fruit="apples pears peaches plums";
2    $fruit =~ /(.*)\s(.*)\s(.*)/;
3    print "$1\n";
4    print "$2\n";
5    print "$3\n";
     print "-" x 30, "\n";
6    $fruit="apples pears peaches plums";
7    $fruit =~ /(.*?)\s(.*?)\s(.*?)\s/; # Turn off greedy quantifier
8    print "$1\n";
9    print "$2\n";
10   print "$3\n";

(Output)
3    apples pears
4    peaches
5    plums
     ------------------------------
8    apples
9    pears
10   peaches
```

解释

1. 将标量 $fruit 赋值为字符串。
2. 将字符串分隔为三个记忆子串，其中每个子串都位于括号内。元字符序列 .* 负责读取零个或多个任意字符。* 将始终匹配可能出现的最长模式，即整个字符串。而在括号外侧还有两个空白字符，它们也必须在字符串中进行匹配。那么，可以保存在 $1 中，并能在字符串中保留两个空格的最大可能的模式是什么？答案自然是 apple pears。
3. 打印 $1 的值。
4. 第一个子串存储在 $1 中。peaches plum 则是原字符串的剩余部分。现在可供匹配且仍有一个空白字符的最大可能的模式（.*）是什么？答案是 peaches。因此，程序会把 peaches 赋值给 $2。打印 $2 的值。
5. 打印第三个子串。plum 是 $3 的内容。
6. 再次将标量 $fruit 赋值为字符串。
7. 本行的贪婪限定符（*）后面带有问号，这就意味着保存的模式将寻找最小数目而不是最大数目的匹配情况。apple 是保存在变量 $1 中的字符数目最少的字符串，pear 是保存在 $2 中的字符数目最少的字符串，而 peaches 则是保存在 $3 中的字母数目最少的字符串。这里必

须提供 \s，否则其最小数目的字符串将是零，因为限定符 * 的本意是零个或多个前面的字符。
8. 打印 $1 的值。
9. 打印 $2 的值。
10. 打印 $3 的值。

关闭捕获。如果用户只希望通过括号分组，对保存在 $1、$2 和 $3 中的替换子串不感兴趣的话，可以使用元字符 ?: 取消捕获子模式。

示例 9.43

```
(In Script)
1   $_="Tom Savage and Dan Savage are brothers.\n";
2   print if /(?:D[a-z]*|T[a-z]*) Savage/;    # Perl will not capture
                                              # the pattern
3   print $1,"\n";      # $1 has no value

(Output)
2   Tom Savage and Dan Savage are brothers.
3   <Nothing is printed>
```

解释

1. 将标量 $_ 赋值为字符串。
2. 当模式位于括号中时，元字符 ?: 能够关闭捕获。本例使用交替模式搜索两个模式。如果搜索成功，就打印 $_ 值。不论找到哪个模式，都必须捕获它并赋值给 $1。
3. 若不使用 ?: 的话，$1 的值就是 Tom，因为它是匹配到的第一个模式。?: 表示“在找到模式时不去保存它”。因此本行没有保存和打印任何内容。

向前向后查找的元字符。字符串中的向前向后查找模式为正则表达式提供了进一步的控制手段。详见表 9-8。

当使用正的向前查找模式时，Perl 会在字符串中向前查找所需模式（?=pattern）。如果找到了该模式，则继续执行正则表达式的模式匹配动作。负的向前查找模式则会向前查看模式（?!pattern）是否存在，如果不存在，这才完成模式匹配。

当使用正的向后查找模式时，Perl 会在字符串中向后查找所需模式（?<=pattern）。如果找到了该模式，则继续执行正则表达式的模式匹配动作。负的向后查找模式则会向后查看模式(?<!pattern）是否存在，如果不存在，这才完成模式匹配。

表 9-8　往返断言

元字符	匹配项
/PATTERN(?=pattern)/	正向前查找
/PATTERN(?!pattern)/	负向前查找
(?<=pattern)/PATTERN/	正向后查找
(?<!pattern)/PATTERN/	负向后查找

示例 9.44

```
(The Script)
```

本 PDF 书签由 龙睿·LoRui 制作。

www.LoRui.com

来看看这些东东……

- ✧ Yahoo!疯了！
- ✧ HP笔记本激活Windows 7

域名转让：

- ✓ DaNanPing.com（大南平）
- ✓ 15500.net
- ✓ Gn7c.com
- ✓ DaLongNan.com（大陇南、大龙南）
- ✓ 51perl.com（无忧 Perl）
- ✓ LoRui.com

```
    # A positive look ahead
1   $string="I love chocolate cake and chocolate ice cream.";
2   $string =~ s/chocolate(?= ice)/vanilla/;
3   print "$string\n";

4   $string="Tomorrow night Tom Savage and Tommy Johnson will leave
             for vacation.";
5   $string =~ s/Tom(?=my)/Jere/g;
6   print "$string\n";
(Output)
3   I love chocolate cake and vanilla ice cream.
6   Tomorrow night Tom Savage and Jeremy Johnson will leave for vacation.
```

解释

1. 标量 $string 中出现了两次 chocolate 模式。其中，chocolate 的第一个具体值后面是字 cake，第二个具体值后面则是 ice。
2. 这是一个正向前查找的示例。模式 chocolate 后面的（?=ice）的意思是：向前检查 ice 是不是下一个模式。如果在 chocolate 之前找到了 ice，则说明匹配成功，并将 chocolate 替换为 vanilla。
3. 完成对第二行的替换操作后，打印新的字符串内容。
4. 将标量 $string 赋值为以 Tom 开头的三个字组成的字符串。
5. 当字符串含有 Tom，且仅当 Tom 后面跟着 my 时，模式才算匹配。如果正向前查找成功的话，则在字符串中将 Tom 替换成 Jere。
6. 完成对第二行的替换操作后，打印新的字符串内容。可见 Tommy 已经替换成了 Jeremy。

示例 9.45

```
(The Script)
    # A negative look ahead
1   while(<DATA>){
2       print if/^\w+\s(?![BC])/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Jon DeLoach
Karen Evich
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 正则表达式的意思是: 在行首位置搜索一个或多个字（\w+），其后是一个空格符（\s），同时向前查找那些既不是 B 又不是 c 的字符。这又称作负向前查找。

示例 9.46

```
(The Script)
    # A positive look behind
1   $string="I love chocolate cake, chocolate milk,
             and chocolate ice cream.";
2   $string =~ s/(?<= chocolate) milk/ candy bars/;
3   print "$string\n";

4   $string="I love coffee, I love tea, I love the boys
             and the boys love me.";
5   $string =~ s/(?<=the boys) love/ don't like/;
6   print "$string\n";

(Output)
3   I love chocolate cake, chocolate candy bars, and chocolate ice cream.
6   I love coffee, I love tea, I love the boys and the boys don't like me.
```

解释

1. 将标量 $string 赋值为含有三个 chocolate 模式具体值的字符串。
2. 括号中的模式又称作正向后查找。Perl 会在字符串中向后查看模式是否出现，如果找到模式 milk 的话，Perl 便会再向后查找字符串，看其前面是不是 chocolate。如果是的话，便把 milk 替换为 candy bars。
3. 在完成替换操作后，打印字符串内容。
4. 本行是另一个正向后查找的示例。Perl 会在字符串中向后查找模式 the boys。如果找到模式，则将正则表达式替换为 don't like。

示例 9.47

```
(The Script)
    # A negative look behind
1   while(<DATA>){
2       print if /(?<!Betty) B[a-z]*/;
    }
    _ _DATA_ _
    Steve Blenheim
    Betty Boop
    Igor Chevsky
    Norma Cord
    Jon DeLoach
    Karen Evich

(Output)
Steve Blenheim
```

解释

1. 特殊文件句柄 DATA 从 _DATA_ 标记后面的文本中获得输入内容。进入 while 循环，读取 _DATA_ 后的第一行内容，并赋值给 $_。每次进入循环体时，将 _DATA_ 后面的一行内容赋值给 $_，直到所有行都处理完毕。
2. 位于括号中的模式又称为负向后查找。Perl 会在字符串中向后查找模式，看它有没有出现。只要其后的模式不是 Betty，便可打印以字母 B 开头并跟有零到多个小写字母 [a-z]* 的行。

9.1.4 tr 或 y 函数

tr 函数⊖能以一一对应的方式将搜索字符串中的字符转译为替换字符串中的相应字符。tr 函数的返回值是替换的字符总数。tr 函数不能解释正则表达式中的元字符，但允许通过散列变量表示某个范围内的字符。这里可用字母 y 来替换 tr。该特性继承自 UNIX，因为后者提供的 sed 实用工具中有一个 y 命令。该命令也能用于转换字符，其效果类似于 UNIX 中的 tr 命令。这也从另一个方面说明了 UNIX 在 Perl 开发过程中的重要作用。

d 选项用于删除搜索字符串。

c 选项用于补全搜索字符串。

s 选项又称作挤压（squeeze）选项。它能把搜索字符串中找到的多个具体值替换成该字符相应的单个具体值（例如，将多个制表符替换为单个制表符）。有关这些修饰符的详细列表请参考表 9-9。

表 9-9 tr 修饰符

修饰符	含 义
d	删除字符
c	补全搜索列表
s	将多个字符挤压（squeeze）为单个字符

格式：

```
tr/search/replacement/
tr/search/replacement/d
tr/search/replacement/c
tr/search/replacement/s
y/search/replacement/      (same as tr; uses same modifiers)
```

示例 9.48

```
(The Input Data)
    Steve Blenheim 101
    Betty Boop 201
    Igor Chevsky 301
    Norma Cord 401
    Jon DeLoach 501
    Karen Evich 601

(Lines from a Script)
1   tr/a-z/A-Z/;print;

(Output)
STEVE BLENHEIM  101
BETTY BOOP  201
IGOR CHEVSKY  301
NORMA CORD  401
JON DELOACH  501
KAREN EVICH  601

2   tr/0-9/:/; print;

(Output)
Steve Blenheim :::
```

⊖ Perl 提供的 tr 函数来自于 UNIX 中的 tr 命令。

```
Betty Boop :::
Igor Chevsky :::
Norma Cord :::
Jon DeLoach :::
Karen Evich :::
3   tr/A-Z/a-c/;print;

(Output)
cteve blenheim 101
betty boop 201
cgor chevsky 301
corma cord 401
con cecoach 501
caren cvich 601

4   tr/ /#/; print;

(Output)
Steve#Blenheim#101
Betty#Boop#201
Igor#Chevsky#301
Norma#Cord#401
Jon#DeLoach#501
Karen#Evich#601

5   y/A-Z/a-z/;print;

(Output)
steve blenheim 101
betty boop 201
igor chevsky 301
norma cord 401
jon deloach 501
karen evich 601
```

解释

1. tr 函数在搜索字符串的每个字符与替换字符串的每个字符之间进行一一对应。本行将把小写字母转换为相应的大写字母。
2. 将每个数字都转换为冒号。
3. 这里的转换过程是杂乱无章的。这是因为搜索端的字符数目比替换端的多，造成从 D 到 Z 的所有大写字母都会替换为 c。
4. 将每个空格符都替换成磅号（#）。
5. 这里的 y 和 tr 意义相同。把每个大写字母转换成相应的小写字母。

tr 删除选项。选项 d 能从搜索字符串里删除那些没有出现在替换字符串中的字符。

示例 9.49

```
1   tr/ //; print;

(Output)
1 Steve Blenheim
2 Betty Boop
```

```
3 Igor Chevsky
4 Norma Cord
5 Jon DeLoach
6 Karen Evich

2   tr/ //d;print;

(Output)
1SteveBlenheim
2BettyBoop
3IgorChevsky
4NormaCord
5JonDeLoach
6KarenEvich
```

解释

1. 在本例中，并没有像使用 sed 或 vi 那样发生转换。
2. 这里需要在使用 tr 函数时提供 d 选项，以便删除每个空格符。

tr 补全选项。选项 c（complement）负责补全搜索字符串；也就是把没有出现在该字符串中的字符转换为替换字符串里的相应字符。

示例 9.50

```
1   tr/0-9/*/; print;

(Output)
* Steve Blenheim
* Betty Boop
* Igor Chevsky
* Norma Cord
* Jon DeLoach
* Karen Evich

2   tr/0-9/*/c; print;

(Output)
1***************2************3**************4************5****
*********6*************
```

解释

1. 本行没有使用 c 选项，因此 tr 函数会把每个数字转换为星号（*）。
2. 本行用到了 c 选项，因此 tr 会把每个不是数字的字符转换为星号（*）。换行符也不例外。

tr 挤压选项。选项 s（squeeze）负责把所有重复的字符替换为相应的单个字符。

示例 9.51

```
(The Text File)
1   while (<DATA>){
        tr/:/:/s;
        print;
    {
    _ _DATA_ _
    1:::Steve Blenheim
```

```
     2::Betty Boop
     3:Igor Chevsky
     4:Norma Cord
     5:::::Jon DeLoach
     6:::Karen Evich


(Output)
1:Steve Blenheim
2:Betty Boop
3:Igor Chevsky
4:Norma Cord
5:Jon DeLoach
6:Karen Evich
```

解释

将多个冒号转换（挤压）为单个冒号。

9.2 Unicode

Unicode 编码为每个字符都规定了惟一的数字标识，以便在不同应用程序、语言和系统平台之间保障信息的一致性。

随着 Internet 的出现，经常需要在全世界范围内从一个 Web 站点向另一个 Web 站点传输数据，并要求不破坏数据内容。这时，ASCII 编码显然就不够用了。ASCII 字符编码序列只能提供 256 个（即一个字节）字符，很难适应汉语或日语之类的情况，因为这些语言往往拥有规模上千的字符集合。

Unicode 标准通过创建新的字符集（称为 UTF8 和 UTF16）来尝试解决上述问题。以 UTF8 为例，它支持 2 个字节，能提供 65536 个字符，并为每个字符提供惟一的编号。如需消除编码的多义性，则应当让给定的 16 位编码值始终指向同一个字符，这样便能实现一致的文本排序、搜索、显示和编辑功能。根据 Unicode 协会[⊖]的资料，Unicode 编码具有支持超过一百万个字符的能力，足以满足全世界各种语言的编码需求。此外，它对所有符号都能一视同仁，因此用户无需转义序列或控制码就能方便地访问所有的字符。

Perl 与 Unicode

Perl 5.6 最大的变化就是提供了对 UTF8 编码的支持。在默认情况下，Perl 内部都通过 Unicode 来表示字符串，而相关的内建函数（如 length、reverse、sort、tr 等）都是以字符为单位（而非以字节为单位）工作的。Perl 提供了两种新的编译指示符，能够打开或者关闭 Unicode 编码方式。其中，utf8 编译指示符能开启 Unicode 编码，并载入所需字符编码表；而 bytes 编译指示符则负责启用传统的字节编码方式，每次只处理一个字节。

在打开 utf8 编码方式时，用户可使用 \x{N} 标记来以 Unicode 编码表示字符，其中 N 是十六进制的字符编码，如 \x{395}。

⊖ Unicode 协会是一个非盈利性组织，负责开发、扩展和促进 Unicode 编码标准的使用。有关 Unicode 编码及其协会的更多信息，请参考：http://www.unicode.org/unicode/standard/whatisunicode.html。

Unicode 提供了对正则表达式的支持，并能基于 Unicode 的属性值来匹配字符。其中有些属性是 Unicode 标准规定的，还有一些则是由 Perl 规定的。Perl 属性是一系列标准属性的组合体。换而言之，读者现在可使用 \p{IsUpper} 来匹配任何语言中的大写字符。更多信息请参阅 http://www.Perl.com/pub/a/2000/04/whatsnew.html。

表 9-10 列出了所有 Perl 组合字符类。如果 \p 中的 p 是大写的话，则表示相反的含义。譬如，\p{IsASCII} 表示 ASCII 字符，而 \P{IsASCII} 则表示非 ASCII 字符。

表 9-10　uft8 组合字符类

uft8 属性	含　义
\p{IsASCII}	ASCII 字符
\p{Cntrl}	控制字符
\p{IsDigit}	0 到 9 之间的数字
\p{IsGraph}	字母数字或者标点符号
\p{IsLower}	小写字母
\p{IsPrint}	字母、数字、标点符号或空格
\p{IsPunct}	标点符号
\p{IsSpace}	空白字符
\p{IsUpper}	大写字母
\p{IsWord}	字母数字字符或者下划线
\p{IsXDigit}	十六进制数字

示例 9.52

```
1  use utf8;
2  $chr=11;
3  print "$chr is a digit.\n"if $chr =~ /\p{IsDigit}/;
4  $chr = "junk";
5  print "$chr is not a digit.\n"if $chr =~ /\P{IsDigit}/;
6  print "$chr is not a control character.\n"if $chr =
       ~ /\P{IsCntrl}/;

(Output)
3   11 is a digit.
5   junk is not a digit.
6   junk is not a control character.
```

解释

1. 使用 utf8 编译指示符打开 Unicode 设置。
2. 将一个数字赋值给标量 $chr。
3. 借助 Perl 提供的 Unicode 属性 IsDigit 判断它是不是一个介于 0 到 9 之间的数字，其效果类似于 [0-9]。
4. 将标量 $chr 赋值为字符串 junk。
5. 注意这里的 \p 现在成了 \P，使得转义序列的意思变成了“非数字”。其效果类似于 [^0-9]。

由于本行中的 junk 不是数字，因此该条件值为真（true）。
6. 反转 junk 得到的值不是控制字符。

9.3 读者应当学到的知识

1. 元字符的用途是什么？
2. 什么是字符类？
3. 什么是“贪婪（greedy）”元字符？
4. 什么是锚定（anchoring）元字符？
5. 如何搜索实际的点号？
6. 什么是捕获（capturing）？如何关闭它？
7. 什么是分组（grouping）？
8. 字符类和交替（alternative）有什么不同？
9. 如何查找一个或多个数字字符？
10. 如何查找零个或一个数字字符？
11. 什么是元符号（metasymbol）？
12. 在使用 tr 函数时，使用“挤压（squeeze）”选项的目的是什么？
13. 什么是 utf8 ？

9.4 下章简介

在下一章中，本书将讨论 Perl 是如何处理文件的，包括如何打开文件、读取文件、写入文件、追加文件内容以及如何关闭文件。读者将了解“die”是怎样工作的，还会学习如何在文件中找到指定位置、如何从文件末尾绕回到文件头、如何标记下一次读取操作的位置。此外，读者还将学习如何检查文件的可读性、可写性和可执行性等。本书将讨论管道（pipe）的相关知识，包括 Perl 是如何把输出内容发送到管道的，以及如何从管道中读取输入内容。最后，读者将学到如何在命令行中通过 ARGV 向 Perl 脚本传递参数。

练习 9

（请在华章网站（www.hzbook.com）上找到如下示例文件。）

```
Tommy Savage:408-724-0140:1222 Oxbow Court, Sunnyvale,CA
94087:5/19/66:34200
Lesle Kerstin:408-456-1234:4 Harvard Square, Boston, MA
02133:4/22/62:52600
JonDeLoach:408-253-3122:123 Park St., San Jose, CA 94086:7/25/53:85100
Ephram Hardy:293-259-5395:235 Carlton Lane, Joliet, IL
73858:8/12/20:56700
etty Boop:245-836-8357:635 Cutesy Lane, Hollywood, CA
91464:6/23/23:14500
Wilhelm Kopf:846-836-2837:6937 Ware Road, Milton, PA
93756:9/21/46:43500
```

```
Norma Corder:397-857-2735:74 Pine Street, Dearborn, MI
23874:3/28/45:245700
James Ikeda:834-938-8376:23445 Aster Ave., Allentown, NJ
83745:12/1/38:45000
Lori Gortz:327-832-5728:3465 Mirlo Street, Peabody, MA
34756:10/2/65:35200
Barbara Kerz:385-573-8326:832 Ponce Drive, Gary, IN
83756:12/15/46:268500
```

1. 打印 Norma 所在的城市和州名。
2. 为每个人增加 $250.00。
3. 计算 Lori 的年龄。
4. 打印从第二行到第六行的所有内容（$. 变量中含有当前行号）。
5. 打印区号为 408 的人名与电话号码。
6. 打印第 3、4、5 行中的人名与工资。
7. 在第三行后面打印一行星号（*）。
8. 将 CA 改名为 California。
9. 打印文件内容，并在最后一行后面加上一行星号。
10. 打印在三月份出生的人的名字。
11. 打印所有不含 Karen 的行。
12. 打印末尾正好是 5 个连续数字的行。
13. 打印文件，并将姓和名反转。

第 10 章　获得文件句柄

10.1　用户定义文件句柄

如果需要处理文本的话，通常就会对文件执行打开、关闭、读取和写入等操作。在 Perl 中，一般都使用文件句柄来访问系统文件。

文件句柄（file handler）是文件、设备、管道或套接字（socket）的名字。在第 4 章“获得打印句柄”中，本书已经讨论过三个默认的文件句柄，即 STDIN、STDOUT 和 STDERR。除此之外，Perl 还允许用户创建自定义的文件句柄，以便对文件、设备、管道或套接字执行输入或输出操作。用户还可把文件句柄关联到系统文件[⊖]上，并通过文件句柄访问这些系统文件。

10.1.1　打开文件：open 函数

open 函数允许用户命名一个文件句柄以及附加在该句柄上的文件。用户可以通过读方式、写方式或追加方式（即读写方式）打开文件，也可打开文件在进程间来回传输数据。如果打开成功，open 函数便返回非零结果；如果失败的话，则返回未定义值。与标量、数组和标签一样，文件句柄也拥有自己的命名空间，因此它们不会与保留字混淆。Perl Wizard 推荐使用全部大写字母的文件句柄名（详见附录 A 中的“open 函数”）。

如需在 Win32 平台上打开文本文件，则在读取磁盘上的文本文件时，\r\n（回车和换行符）会自动切换为 \n，而 ^Z 字符则读作文件末尾标识符（即 EOF）。本章下面将要介绍的函数都能顺利地处理文本，但在处理二进制文件时却会产生问题（详见 10.1.4“Win32 二进制文件”）。

10.1.2　打开文件读取

下面的示例说明了如何以读方式打开一个文件。尽管本例针对的是 UNIX 文件，但它和 Windows 及 Mac OS 等操作系统上的工作方式并没有差别。

格式：

```
open(FILEHANDLE, "FILENAME");
open(FILEHANDLE, "<FILENAME");
```

⊖ 所谓系统文件，指的是 UNIX、Win32、Macintosh 自带的文件，位于系统磁盘上。

```
2   open(FILEHANDLE);
3   open FILEHANDLE;
```

示例 10.1

```
1   open(MYHANDLE, "myfile");
2   open (FH, "</etc/passwd");
3   open (MYHANDLE);
```

解释

1. open 函数创建一个文件句柄 MYHANDLE，并将其附加给系统文件 myfile。以读方式打开该文件。由于没有为 myfile 文件指定全路径，因此该文件必须位于当前目录下，并提供读权限。这样才能成功地以读方式打开它。
2. open 函数创建一个文件句柄 FH，并将它附加给系统文件 /etc/passwd。以读方式打开该文件，不过这次是通过 < 符号来说明读方式的。< 符号在这里并不是必需的，但它有助于说明这是一个读操作。本行为 passwd 文件指定了全路径。
3. 如果省略了 FILENAME，则文件句柄的名字将与前面定义的标量名相同。本行把实际文件的名字赋值给了标量型变量。在本例中，可以将文件名定义为：

```
$MYHANDLE="myfile";
open(MYHANDLE);
```

Open 函数创建一个文件句柄 FH，并将它附加给标量 $MYHANDLE 的值。其作用与第一个例子相同。本行的括号是可有可无的。

关闭文件句柄。close 函数负责关闭附加给 FILEHANDLE 文件句柄的文件、管道、套接字或设备。一旦程序打开了 FILEHANDLE 文件句柄，则直到脚本结束或再次调用 open 函数（后一次 open 调用会自动关闭前一次 open 的句柄）之前，该句柄都是处于打开状态的。如果没有显式地关闭文件，则通过上述方式打开文件后，行计数器变量 $. 将不会重置。关闭管道文件将引起相关进程陷入等待状态，直到管道操作完成并在 $! 变量中报告完成状态（详见“die 函数”一节）为止。综上所述，在使用完文件和句柄后，必须及时关闭。

格式：

```
close (FILEHANDLE);
close FILEHANDLE;
```

示例 10.2

```
open(INFILE, "datebook");
close(INFILE);
```

解释

关闭用户定义的文件句柄 INFILE。

die 函数。在下面的示例中，如果调用 open 函数失败，程序便会调用 die 函数。如果 Perl 脚本无法打开指定文件，则可通过 die 函数退出脚本，并把错误信息打印到 STDERR（通常就是终端屏幕）。

譬如，在 shell 或 MS-DOS 提示符下输入：

```
cat junk (UNIX)
```

或

```
type junk  (DOS)
```

并且已知这个 junk 文件并不存在。则在屏幕上将出现如下错误信息：

```
cat: junk: No such file or directory    (UNIX "cat" command)
The system cannot find the file specified.   (Windows "type" command)
```

Perl 特殊变量 $! 中含有系统错误信息，这些错误信息往往是由于不能成功打开文件或在执行系统实用程序时出错而造成的（详见“错误处理”一节）。在使用 die 函数时，该特殊变量能在继续执行脚本之前获得文件句柄出错的具体原因（详见示例 12.10 中讨论的 Carp.pm 模块）。

示例 10.3

```
(Line from Script)
1   open(MYHANDLE, "/etc/password) || die  "Can't open: $!\n";
2   open(MYHANDLE, "/etc/password) or die  "Can't open: $!\n";

(Output)
1   Can't open: No such file or directory
(Line from Script)
3   open(MYHANDLE,   "/etc/password") || die  "Can't open: ";

(Output)
3   Can't open: No such file or directory at ./handle line 3.
```

解释

1. 当尝试打开文件 /etc/password 时，open 操作失败（因为路径应当是 /etc/passwd）。当左边的操作数失败（即为假）时，短路运算符会继续执行其右侧的操作数。因此本行将继续执行 die 运算符。这里将打印字符串 Can’t Open，随后是系统错误信息“No such file or directory”。换行符 \n 中止了 die 函数的进一步输出。在退出程序时，上述所有 die 输出都会发送给 STDERR。
2. 考虑到代码可读性，这里使用 or 运算符，而不是 ||。
3. 本例与第一个例子几乎完全一样，只不过它不含 \n 换行符。省略 \n 将使得 die 函数把另一个字符串追加到输出内容中，以表明脚本中发生错误的代码行号。

从文件句柄中读取。 示例 10.4 以读方式打开了文件 datebook，并将读到的各行内容依次赋值给标量 $_。这是一个默认的变量，含有读取到的直到文件末尾的所有内容。

示例 10.4

```
(The Text File: datebook)
    Steve Blenheim
    Betty Boop
    Lori Gortz
    Sir Lancelot
    Norma Cord
    Jon DeLoach
    Karen Evich

-----------------------------------------------------------------
```

```
(The Script)
    #!/usr/bin/perl
    # Open a file with a filehandle
1   open(FILE, "datebook") || die "Can't open datebook: $!\n";
2   while(<FILE>) {
3       print if /Sir Lancelot/;
4   }
5   close(FILE);

(Output)
3   Sir Lancelot
```

解释

1. open 函数创建一个文件句柄（以读方式打开），并将其附加到系统文件 datebook 上。由于该文件不存在，因此 die 函数将打印以下内容到屏幕上：Can’t open datebook: No such file or directory。
2. while 循环中的表达式是文件句柄 FILE，它位于角括号之间。角括号是表示读取操作的运算符（它们不是文件句柄名称的一部分）。当循环开始时，从文件句柄 FILE 中读取其第一行内容，并将它存入标量 $_ 中（请记住，$_ 标量含有从文件中读取到的每一行内容）。如果尚未达到文件末尾，循环就将继续读取文件的每一行内容，即执行语句 3 和 4，直到文件结束。
3. 默认输入标量 $_ 用于隐式地保存从文件句柄中读到的当前输入行。如果该行中含有正则表达式 Sir Lancelot，则将其（位于 $_ 中）打印到 STDOUT。每次循环时，都会把读到的下一行文件内容保存到 $_，并查看其内容。
4. 位于末尾位置的花括号表明循环体的结束位置。当程序到达这一行时，就将控制流返回到循环顶部（即第 2 行处），然后继续从文件中读取下一行内容。该过程一直持续，直到读完文件中的所有行。
5. 在循环读取完毕整个文件后，通过文件句柄的关闭操作来关闭该文件。

示例 10.5

```
(The Text File: datebook)
    Steve Blenheim
    Betty Boop
    Lori Gortz
    Sir Lancelot
    Norma Cord
    Jon DeLoach
    Karen Evich

-----------------------------------------------------------------

(The Script)
    #!/usr/bin/perl
    # Open a file with a filehandle
    open(FILE, "datebook") || die "Can't open datebook: $!\n";
    while($line = <FILE>) {
        print "$line" if $line =~ /^Lori/;
    }
    close(FILE);

(Output)
    Lori Gortz
```

解释

1. 以读方式打开文件 datebook。
2. 进入 while 循环，从文件中读取一行内容，并将它保存到标量 $line 中。
3. 如果该行内容含有模式 Lori，且该模式位于行首的话，打印标量 $line 的值。
4. 当到达右侧角括号时，将控制流跳转到第 2 行，以便读取文件的下一行内容。在读完文件的所有行后，退出该循环。
5. 通过关闭文件句柄的方式关闭该文件。

示例 10.6

```
(The Text File: datebook)
    Steve Blenheim
    Betty Boop
    Lori Gortz
    Sir Lancelot
    Norma Cord
    Jon DeLoach
    Karen Evich

----------------------------------------------------------------

(The Script)
    #!/usr/bin/perl
    # Open a file with a filehandle
1   open(FILE, "<datebook") || die "Can't open datebook: $!\n";
2   @lines = <FILE>;
3   print @lines;       # Contents of the entire file are printed
4   print "\nThe datebook file contains ", $#lines + 1,
          " lines of text.\n";
5   close(FILE);

(Output)
The datebook file contains 7 lines of text.
```

解释

1. 以读方式打开文件 datebook（本行的读取运算符 < 是可有可无的）。
2. 通过文件句柄读取文件中的所有行，并全部赋值给数组 @lines。文件的每一行内容都成为数组的一个元素，并以换行符结尾。
3. 打印数组 @lines 的内容。
4. 变量 $#lines 的值是数组中前一个子脚本的编号。只需把 $#lines 加上 1，便可得到该元素（行）的编号。偏移量 -1（即 $lines[-1]）表示最后一行。

10.1.3 打开文件写入

当以写方式打开一个文件时，如果该文件不存在，程序就会创建它；如果该文件已存在，则它必须对用户提供写权限。如果文件存在的话，用户可以修改其内容。这里同样使用文件句柄来访问系统文件。

格式：

```
open(FILEHANDLE, ">FILENAME)";
```

示例 10.7

```
open(MYOUTPUT, ">temp");
```

解释

通过用户定义的 MYOUTPUT 文件句柄把内容输出到文件 temp 中。与使用 shell 时一样，这里也要借助重定向符号把输出内容从默认文件句柄 STDOUT 重定向到文件 temp 中。

示例 10.8

```
(The Script)
    #!/usr/bin/perl
    # Write to a file with a filehandle. Scriptname: file.handle
1   $file="/home/jody/ellie/perl/newfile";
2   open(HANDOUT, ">$file") || die "Can't open newfile: $!\n";

3   print HANDOUT "hello world.\n";
4   print HANDOUT "hello world again.\n";

(At the Command Line)
5   $ perl file.handle
6   $ cat newfile

(Output)
3   hello world.
4   hello world, again.
```

解释

1. 将标量型变量 $file 赋值为 UNIX 文件 newfile 的全路径名称。本行将通过文件句柄把输出内容重定向到该变量指向的 UNIX 文件中。在 Windows 上，本例的工作方式基本相同，但由于需要使用反斜杠作为目录分隔符，因此必须用单引号围住路径，或者提供双反斜杠字符，例如：C:\\home\\ellie\\testing。
2. 用户定义了一个名叫 HANDOUT 的文件句柄，从而将输出内容的传送目的地从默认的 STDOUT 重定向为它所代表的 newfile 文件。> 符号表示：如果该文件不存在，则创建它，并以写方式打开它；如果该文件存在，则打开它，然后覆盖文件原有内容。因此在执行时应当慎之又慎。
3. print 函数将输出内容重定向到文件句柄 HANDOUT 而非屏幕。本行程序将通过文件句柄 HANDOUT 将字符串 hello world 写入到文件 newfile 中去。此后，newfile 文件将一直保持在打开状态，直到代码显式地关闭它，或者退出 Perl 脚本。
4. print 函数将输出内容重定向到文件句柄 HANDOUT 而非屏幕。本行程序将通过文件句柄 HANDOUT 将字符串 hello world 写入到文件 newfile 中。操作系统将跟踪上一次写入文件的位置，并把下一行输出内容发送到文件末尾处。
5. 执行脚本，将输出内容重定向到 newfile 文件内。
6. 打印文件 newfile 的内容。

10.1.4 Win32 二进制文件

Win32 中的文件分为两种：文本文件和二进制文件。如果在文本文件中发现 ^Z 字符，则可能会造成程序退出或者碰到换行符转换方面的问题。在读写 Win32 文件时，用户可使用 binmode 函数避免上述问题。binmode 函数负责在读写文件时指定模式，如二进制（raw）文件或文本文

件。如果用户没有提供合乎规范的参数的话，其默认模式将设定为“raw”。可选的规范包括：:raw、:crlf、:text、:utf8 以及 :latin1 等。

格式：

```
binmode FILEHANDLE
binmode FILEHANDLE, DISCIPLINE
```

示例 10.9

```
# This script copies one binary file to another.
# Note its use of binmode to set the mode of the filehandle.

1   $infile="statsbar.gif";
2   open( INFILE, "<$infile" );
3   open( OUTFILE, ">outfile.gif" );

4   binmode( INFILE );      # Crucial for binary files!

5   binmode( OUTFILE );
    # binmode should be called after open() but before any I/O
    # is done on the filehandle.
6   while ( read( INFILE, $buffer, 1024 ) ) {
7       print OUTFILE $buffer;
    }

8   close( INFILE );
    close( OUTFILE );
```

解释

1. 将标量 $infile 赋值为 .gif 文件名。
2. 打开文件 statsbar.gif，以读方式打开该文件，并附加给文件句柄 INFILE。
3. 以写方式打开文件 outfile.gif，并追加给文件句柄 OUTFILE。
4. 使用 binmode 函数，以二进制方式读取文件内容。
5. 使用 binmode 函数，以二进制方式写入文件内容。
6. read 函数每次读取 1024 字节内容，并将读取到的输入内容保存到标量 $buffer 中。
7. 在读取到 1024 字节内容后，将它们发送至输出文件。
8. 关闭上述两个文件句柄。本程序能把一个二进制文件内容拷贝给另一个二进制文件。

10.1.5 打开文件追加

在以追加方式打开一个文件时，如果该文件不存在，则创建文件；如果该文件已经存在，则必须提供写权限。如果文件存在的话，原文件的内容将保持不变，程序将把输出内容添加到原文件的末尾。同样，在访问文件时，程序必须使用文件句柄，而不是真实的文件名。

格式：

```
open(FILEHANDLE, ">> FILENAME");
```

示例 10.10

```
open(APPEND, ">> temp");
```

解释

使用用户定义的文件句柄，将输出内容添加到文件 temp 末尾。与使用 shell 一样，本行的重定向符负责把输出内容从默认的 STDOUT 文件句柄重定向到文件 temp 上。

示例 10.11

```
(The Text File)
$ cat newfile
hello world.
hello world, again.

(The Script)
    #!/usr/bin/perl
1   open(HANDLE, ">>newfile") ||
            die print "Can't open newfile: $!\n";
2   print HANDLE "Just appended \"hello world\"
            to the end of newfile.\n";

(Output)
$ cat newfile
hello world.
hello world, again.
Just appended "hello world" to the end of newfile.
```

解释

1. 使用用户定义的文件句柄，将输出内容添加到文件 newfile 中。与使用 shell 时一样，这里也通过重定向符号把输出内容从默认的 STDOUT 文件句柄重定向并添加到文件 newfile 中。如果无法打开文件（譬如该文件关闭了写权限）的话，则通过 die 函数打印出错消息：Can’t open newfile: Permission denied.，然后退出脚本。
2. print 函数负责将输出内容发送到文件句柄 HANDLE，而不是发送给终端屏幕。本示例将通过文件句柄 HANDLE 将字符串 Just appended “hello world” to the end of newfile 添加到文件 newfile 的末尾。

10.1.6 select 函数

select 函数能把默认输出设置为用户定义的 HANDLE 文件句柄，并返回前面选定的文件句柄。这样，程序便能把所有内容打印到指定的文件句柄中。

示例 10.12

```
(The Script)
    #! /usr/bin/perl
1   open (FILEOUT,">newfile")  || die "Can't open newfile: $!\n";
2   select(FILEOUT);       # Select the new filehandle for output
3   open (DB, "<datebook") || die "Can't open datebook: $!\n";
    while(<DB>) {
4      print ;             # Output goes to FILEOUT, i.e., newfile
    }
5   select(STDOUT);        # Send output back to the screen
    print "Good-bye.\n";   # Output goes to the screen
```

解释

1. 以写方式打开文件 newfile，并将它指派给文件句柄 FILEOUT。
2. select 函数将当前输出内容的默认文件句柄设置为 FILEOUT。select 函数返回的内容是该函数为了选择 FILEOUT 文件句柄而关闭的那个文件句柄名，也就是下面以读方式打开的文件句柄。
3. 以读方式打开 DB 文件句柄。
4. 将 DB 中的每一行依次读入标量 $_，然后打印当前选择的 FILEOUT 句柄的内容。请注意，这里无需对文件句柄命名。
5. 通过选择 STDOUT 文件句柄，将程序的剩余输出内容重定向到终端屏幕。

10.1.7 使用 flock 为文件加锁

如要避免两个程序同时写同一个文件，用户可以先为该文件加锁，然后由一个程序单独访问它，并在使用完毕后再对文件进行解锁。flock 函数含有两个参数：文件句柄，以及文件锁操作。表 10-1 中列举了所有的文件锁操作[1]。

表 10-1 文件锁操作

操作名	操作	具体工作
lock_sh	1	创建共享锁
lock_ex	2	创建排他锁
lock_nb	4	创建非阻塞锁
lock_un	8	解除当前锁

如要获得共享锁，文件必需能提供读权限；而在获得排他锁时则要求文件提供写权限。一般而言，当使用上述选项 1 和选项 2 时，调用者请求的文件将被锁住（等待），直到解锁为止。如果对文件句柄加上了非阻塞锁，则在请求带锁文件时程序会立刻抛出错误信息[2]。

示例 10.13

```
    #!/bin/perl
    # Program that uses file locking -- UNIX
1   $LOCK_EX = 2;
2   $LOCK_UN = 8;

3   print "Adding an entry to the datafile.\n";
    print "Enter the name: ";
    chomp($name=<STDIN>);
    print "Enter the address: ";
    chomp($address=<STDIN>);

4   open(DB, ">>datafile") || die "Can't open: $!\n";
```

[1] 在非 UNIX 系统上，这些文件锁操作可能是无效的。

[2] 如果是从网络系统访问文件的话，加锁操作就可能是无效的。

```
5   flock(DB, $LOCK_EX) || die ;          # Lock the file
6   print DB "$name:$address\n";
7   flock(DB, $LOCK_UN) || die;           # Unlock the file
```

解释

1. 本行通过 flock 函数对文件加锁，并将所用操作赋值给标量。在创建成功排他锁之前，本行操作将一直处于阻塞（等待）状态。
2. 该操作告诉 flock 函数何时对文件解锁，以便其他人写该文件。
3. 要求用户输入信息，并以此更新文件。该信息将追加到文件末尾。
4. 以追加方式打开文件句柄。
5. flock 函数负责在文件上设置一个排他锁。
6. 将数据追加到文件末尾。
7. 数据追加完毕后，对文件解锁，使得其他人也能访问该文件。

10.1.8 seek 和 tell 函数

seek 函数。负责以随机的方式访问文件。seek 函数等效于 C 语言中的标准 I/O 函数 fseek。seek 函数允许用户在文件中直接跳转到指定内容所在的位置，而不是关闭文件再打开它。如果成功的话，seek 函数将返回 1，否则就返回 0。

格式：

```
seek(FILEHANDLE, BYTEOFFSET, FILEPOSITION);
```

seek 函数能在文件中设定位置值，其中文件第一个字节位置为 0。具体位置包括：

0 ＝ 文件开头位置

1 ＝ 文件中的当前位置

2 ＝ 文件末尾位置

偏移量是从起点位置到文件当前位置的字节数。其中正的偏移量能在文件内向前移动位置；而负偏移量则用于在文件中基于位置 1 或位置 2 向后移动。

od 命令用于查看文件中字符的存储方式。该文件是基于 Win32 系统创建的；而在 UNIX 系统上，这里的进纸 / 换行符将合并为一个字符 \n。

```
$ od -c db
0000000000   S   t   e   v   e       B   l   e   n   h   e   i   m  \r  \n
0000000020   B   e   t   t   y       B   o   o   p  \r  \n   L   o   r   i
0000000040   G   o   r   t   z  \r  \n   S   i   r       L   a   n   c
0000000060   e   l   o   t  \r  \n   N   o   r   m   a       C   o   r   d
0000000100  \r  \n   J   o   n       D   e   L   o   a   c   h  \r  \n   K
0000000120   a   r   e   n       E   v   i   c   h  \r  \n
0000000134
```

示例 10.14

```
(The Text File: db)
Steve Blenheim
Betty Boop
```

```
Lori Gortz
Sir Lancelot
Norma Cord
Jon DeLoach
Karen Evich

-----------------------------------------------------------------

(The Script)
    # Example using the seek function
1   open(FH,"db") or die "Can't open: $!\n";
2   while($line=<FH>){        # Loop through the whole file
3       if ($line =~ /Lori/) { print "--$line--\n";}
     }
4   seek(FH,0,0);             # Start at the beginning of the file
5   while(<FH>) {
6      print if /Steve/;
    }

(Output)
3   --Lori Gortz--
6   Steve Blenheim
```

解释

1. 将 db 文件指派给文件句柄 FH，并以读方式打开该文件。
2. 循环遍历文件内容，依次将文件中的每一行赋值给标量 $line。
3. 如果 $line 中含有 Lori，则执行 print 语句。
4. seek 函数将把文件指针指向文件开头（位置 0），并从字节 0（即第一个字节）开始读取文件内容。如果这里不通过 seek 函数回到文件开头的话，程序就必须先用 close 函数显式地关闭文件句柄。
5. 从文件顶部开始进入循环。从文件句柄中读取文件的第一行内容，并赋值给标量 $_，后者是默认的行存储变量。
6. 如果在 $_ 中找到模式 Steve 的话，打印该行内容。

示例 10.15

```
(The Text File: db)
Steve Blenheim
Betty Boop
Lori Gortz
Sir Lancelot
Norma Cord
Jon DeLoach
Karen Evich

-----------------------------------------------------------------

(The Script)
1   open(FH, "db") or die "Can't open datebook: $!\n";
2   while(<FH>){
3       last if /Norma/;    # This is the last line that
                            # will be processed
    }
4   seek(FH,0,1) or die;    # Seeking from the current position
5   $line=<FH>;             # This is where the read starts again
```

```
6   print "$line";
7   close FH;

(Output)
6   Jon DeLoach
```

解释

1. 通过文件句柄 FH 以读方式打开文件 db。
2. 进入 while 循环。从文件中读取一行内容，并赋值给 $_。
3. 如果发现含有模式 Norma 的行，last 函数将导致程序退出循环。
4. seek 函数将把文件指针重新定位到字节位置 0。这是在该文件中执行下一次读操作的位置，位置 1：即含有 Norma 的行之后一行。该字节位置既可以是正值，也可以是负值。
5. 从文件 db 中读取一行内容，并赋值给标量 $line。这里读到的行是在 last 函数退出循环之后所读取的。
6. 打印标量 $line 的值。

示例 10.16

```
(The Script)
1   open(FH, "db") or die "Can't open datebook: $!\n";
2   seek(FH,-13,2) or die;
3   while(<FH>){
4       print;
    }

(Output)
4   Karen Evich
```

解释

1. 通过文件句柄 FH 以读方式打开文件 db。
2. seek 函数将从文件末尾（位置 2）开始向后退 13 个字节。这里的换行标志（\r\n）虽然不可见，但也占据了本行最后两个字节（Windows）。
3. 进入 while 循环，依次从文件句柄中读取每一行内容。
4. 打印每一行内容。从文件末尾后退 13 个字节，打印 Karen Evich。请留意 od -c 命令的输出内容，即从文件末尾向后数 13 个字符。

```
0000000000   S   t   e   v   e       B   l   e   n   h   e   i   m  \r  \n
0000000020   B   e   t   t   y       B   o   o   p  \r  \n   L   o   r   i
0000000040       G   o   r   t   z  \r  \n   S   i   r       L   a   n   c
0000000060   e   l   o   t  \r  \n   N   o   r   m   a       C   o   r   d
0000000100  \r  \n   J   o   n       D   e   L   o   a   c   h  \r  \n   K
0000000120   a   r   e   n       E   v   i   c   h  \r  \n
0000000134
```

tell 函数。tell 函数能返回文件中当前字节的位置，并能与 seek 函数配合使用，以便在文件中移动到该位置。如果省略了 FILEHANDLE 参数的话，tell 函数将返回文件上一次读取的位置。

格式：

```
tell(FILEHANDLE);
tell;
```

示例 10.17

```
(The Text File: db)
Steve Blenheim
Betty Boop
Lori Gortz
Sir Lancelot
Norma Cord
Jon DeLoach
Karen Evich

------------------------------------------------------------------

(The Script)
    #!/usr/bin/perl
    # Example using the tell function
1   open(FH,"db") || die "Can't open: $!\n";
2   while ($line=<FH>) {       # Loop through the whole file
       chomp($line);
3      if ($line =~ /^Lori/) {
4          $currentpos=tell;
5          print "The current byte position is $currentpos.\n";
6          print "$line\n\n";
       }
    }
7   seek(FH,$currentpos,0);    # Start at the beginning of the file
8   @lines=(<FH>);
9   print @lines;

(Output)
5   The current byte position is 40.
6   Lori Gortz

9   Sir Lancelot
    Norma Cord
    Jon DeLoach
    Karen Evich
```

解释

1. 将 db 文件指定给文件句柄 FH，并以读方式打开该文件。
2. 循环遍历文件内容，依次将文件中的每一行赋值给标量 $line。
3. 如果 $line 中含有 Lori，则进入 if 代码块。
4. 调用 tell 函数，返回文件中的当前字符位置（从字节 0 开始计算）。该位置代表了在处理完含有 Lori 的行之后，再读到的行中第一个字符的位置。
5. 字节值保存在标量 $currentpos 中。打印它的值。字节位置 40 代表了以 Sir Lancelot 开头的行的位置。
6. 打印含有正则表达式 Lori 的行。
7. seek 函数将 FH 文件句柄的文件指针定位到指定的字节偏移量 $currentpos 处，即定位到文件开头的第 40 个字节处。如果这里不用 seek 函数的话，如需从文件顶部开始读取，就必须先关闭文件句柄。
8. 读取从偏移量 40 开始的行，并将其内容保存到数组 @lines 中。
9. 从偏移量 40 开始，打印数组内容。

10.1.9 打开文件读写

表 10-2 读写操作

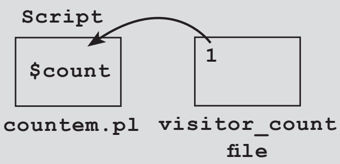
符号	目 的
+<	先读，后写
+>	先写，后读
+>>	先追加内容，然后读

示例 10.18

```
(The Script)
    # Scriptname: countem.pl
    # Open visitor_count for reading first, and then writing
1   open(FH, "+<visitor_count") ||
           die "Can't open visitor_count: $!\n";
2   $count=<FH>;            # Read a number from from the file
3   print "You are visitor number $count.";
4   $count++;
5   seek(FH, 0,0) || die;   # Seek back to the top of the file
6   print FH $count;        # Write the new number to the file
7   close(FH);


(Output)
(First run of countem.pl)
You are visitor number 1.

(Second run of countem.pl)
You are visitor number 2.
```



解释

1. 以先读后写的方式打开文件 visitor_count。读写操作符及其含义请参见表 10.2。如果文件不存在或不可读的话，则通过 die 函数退出脚本，并打印错误信息。
2. 从文件 visitor_coun 中读取一行内容。当脚本第一次执行时，会从 visitor_coun 文件中读到数字 1，并保存至标量 $count。
3. 打印 $count 的值。
4. 将标量 $count 的值递增 1。
5. seek 函数将文件指针移动到文件的开头位置。
6. 将新的 $count 变量值写回 visitor_count 文件。脚本每次执行时，都会用新的 $count 值覆盖文件中原有的数字。
7. 关闭该文件。

示例 10.19

```
(The Script)
    #!/usr/bin/perl
    # Open for writing first, then reading
    print "\n\n";
1   open(FH, "+>joker") || die;
```

```
2   print FH "This line is written to joker.\n";
3   seek(FH,0,0);          # Go to the beginning of the file
4   while(<FH>) {
5       print;             # Reads from joker; the line is in $_
    }

(Output)
5   This line is written to joker.
```

解释

1. 首先以读方式打开文件句柄 FH。也就是说创建文件 joker，或者在该文件已存在时截取它。请读者注意不要混淆 +< 和 +>。
2. 通过文件句柄 FH，将输出内容发送到 joker 文件中。
3. seek 函数将文件指针移动到文件开头位置。
4. 进入 while 循环，通过文件句柄从文件 joker 中读取一行内容，并保存到标量 $_ 中。
5. 在读取完毕后打印每一行（$_）内容，直到文件结束。

10.1.10　打开管道

在使用管道时，用户可以创建从一个程序到另一个程序的数据连接。位于管道左侧的程序会把其输出内容发送到临时的内核缓冲区中，即写入管道。而位于管道另一侧的用户程序则可从上述缓冲区中获取输入内容。下面是一个典型的 UNIX 管道（如图 10-1 所示）：

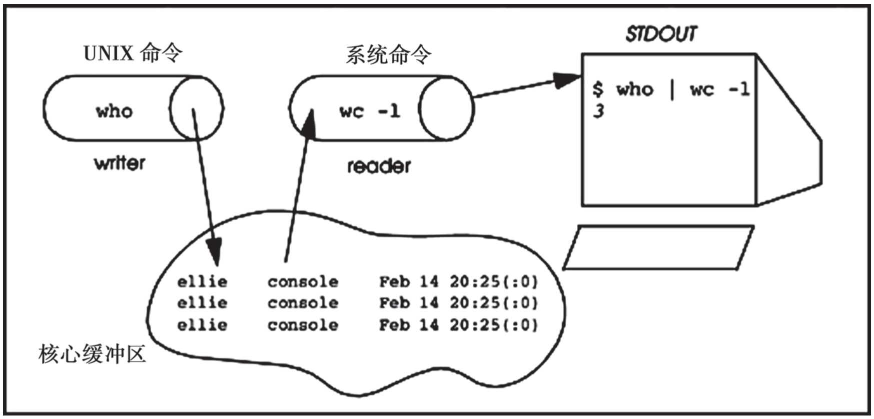


图 10-1　UNIX 管道示例

```
who | wc -l
```

下面则是一个 MS-DOS 管道：

```
dir /B | more
```

上述两行命令的意思都是将 who 命令执行结果发送给 wc 命令。首先由 who 命令将输出内容发送给管道，即写入管道。然后，由 wc 命令从管道中读取其输入，即从管道读（如果 wc 命令不是管道的读取者，它将忽略管道中的内容）。最后把输出内容发送到 STDOUT，即终端屏幕。本命令能够打印出登录系统的人数。如果 Perl 脚本位于管道左侧的话，它就是管道写入方，将会把输出内

容发送到缓冲区中；而倘若 Perl 脚本位于管道右侧的话，它则会从缓冲区中读取输入内容。请读者牢记一点：连接到 Perl 脚本的进程实际上是一个操作系统命令。如果 Perl 程序是运行在 UNIX 或 Linux 操作系统上的话，其命令格式就有可能和 Windows 系统命令格式有所不同。因此，带有管道操作的 Perl 脚本是不能直接在不同系统间互相移植的。

输出过滤器。当使用 open 函数创建文件句柄时，用户可以为它打开输出过滤器（filter），以便将输出内容传送给系统命令（参见图 10-2）。该命令前面带有一个管道符号（|），能在前面示例中代替文件名参数。如此便可把输出内容重定向到该命令处，然后发送给 STDOUT。

格式：

```
open(FILEHANDLE,|COMMAND);
```

示例 10.20

```
(The Script)
    #!/bin/perl
    # Scriptname: outfilter (UNIX)
1   open(MYPIPE, "| wc -w");
2   print MYPIPE "apples pears peaches";
3   close(MYPIPE);

(Output)
3
```

解释

1. 使用用户定义的 MYPIPE 文件句柄，将输出内容从 Perl 脚本重定向到 UNIX 命令 wc -w 处，后者用于计算字符串中的字符数目。
2. print 函数将把字符串 apples pears peaches 转发给带有输出过滤器的 MYPIPE 文件句柄；后者负责将字符串传送给 wc 命令。由于该字符串中含有三个单词，因此本行将把数字 3 打印到屏幕上。
3. 在使用完文件句柄后，应当通过 close 函数关闭它。这样才能保证在退出脚本之前完成所有的命令。如果不关闭文件句柄的话，程序就可能不会正确地清除输出内容。

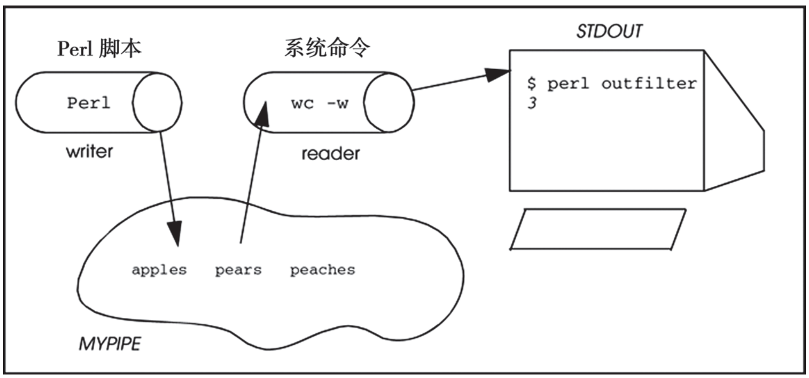


图 10-2　Perl 输出过滤器

示例 10.21

```
(The Script)
1    open(FOO, "| tr '[a-z]' '[A-Z]'");
2    print FOO "hello there\n";
3    close FOO;    # If you don't close FOO, the output may be delayed

(Output)
2    HELLO THERE
```

解释

1. 使用用户定义的 FOO 文件句柄，将 Perl 脚本输出发送给 UNIX 命令 tr。后者能将所有小写字母转换为大写形式。
2. print 函数能将字符串 hello there 转发给带有输出过滤器的 FOO 文件句柄，也就是把该字符串发送给 tr 命令。在过滤之后，程序会把字符串发送到屏幕上。此时，字符串中所有的字符都会转换为大写形式。

示例 10.22

```
(The Text File)
$ cat emp.names
1 Steve Blenheim
2 Betty Boop
3 Igor Chevsky
4 Norma Cord
5 Jon DeLoach
6 Karen Evich

(The Script)
     #!/usr/bin/perl
1    open(FOO, "| sort  +1| tr '[a-z]' '[A-Z]'"); # Open output filter
2    open(DB, "emp.names");          # Open DB for reading
3    while(<DB>){ print FOO ; }
4    close FOO;

(Output)
2    BETTY BOOP
3    IGOR CHEVSKY
5    JON DELOACH
6    KAREN EVICH
4    NORMA CORD
1    STEVE BLENHEIM
```

解释

1. 通过用户定义的文件句柄，将输出内容重定向到 UNIX 命令 sort，然后将 sort 命令的输出内容转发给 tr 命令。其中，sort + 1 命令负责根据第 2 个字段的大小进行排序，各字段由空格隔开。UNIX 命令 tr 则负责将小写字母转换为大写字母。
2. 使用 open 函数创建文件句柄 DB，并将其附加到 UNIX 文件 emp.names 上。
3. while 循环表达式中含有文件句柄 DB，该句柄位于角括号之间，表示读取该文件内容。循环将从 emp.names 文件中读取第一行内容，并保存到标量 $_ 中，然后通过输出过滤器 FOO 转发输入行内容，并打印到屏幕上。在到达文件末尾之前，循环将不断重复执行。请注意，当按照第二个字段对文件进行排序时，其第一列的数字将不参与排序。
4. close 函数关闭文件句柄 FOO。

将过滤器输出转发至文件。　在上述示例中，怎样才能把过滤器的输出内容转发到文件，而不是 STDOUT 呢？用户不能把输出内容同时既发给过滤器又发给文件句柄，但可以把 STDOUT 重定

向到另一个文件句柄去。此后，如果想把 STDOUT 再重定向回屏幕的话，只需首先保存该句柄，然后使用下面这个命令重新打开 STDOUT 文件句柄即可。

```
open (STDOUT," > /dev/tty" );
```

示例 10.23

```
     #!/usr/bin/perl
     # Program to redirect STDOUT from filter to a UNIX file
1    $| = 1;            # Flush buffers
2    $tmpfile = "temp";
3    open(DB, "data") || die qq/Can't open "data": $!\n/;
                                      # Open DB for reading
4    open(SAVED, ">&STDOUT") || die "$!\n";  # Save stdout
5    open(STDOUT, ">$tmpfile" ) || die "Can't open: $!\n";
6    open(SORT, "| sort +1") || die;          # Open output filter
7    while(<DB>){
8       print SORT;   # Output is first sorted and then sent to temp.
9    }
10   close SORT;
11   open(STDOUT, ">&SAVED") || die "Can't open";
12   print "Here we are printing to the screen again.\n";
                        # This output will go to the screen
13   rename("temp","data");
```

解释

1. 变量 $1 保证在每次执行完 print 语句后自动清空输出缓冲区（详见附录 A 中的 autoflush 模块）。
2. 将标量 $tempfile 赋值为 temp，即输出文件名。
3. 以读方式打开 UNIX 中的 data 文件，并附加给文件句柄 DB。
4. 将 STDOUT 内容赋值并转存到另一个文件句柄 SAVED 中。本行将在后台操作相应的文件描述符。
5. 以读方式打开文件 temp，并赋值给文件描述符，其默认值一般是 STDOUT，即终端屏幕。本行将先关闭 STDOUT 文件描述符，然后再为 temp 重新打开。
6. 将输出过滤器指派给 SORT。本行将把 Perl 脚本输出内容转发给 UNIX 的 sort 实用程序。
7. 以读方式打开文件句柄 DB。
8. 在完成排序后，将输出文件句柄发送给 temp 文件。
9. 关闭循环。
10. 关闭输出过滤器。
11. 打开标准输出文件句柄，并将其输出目的地重新设置为终端屏幕。
12. 因为上面已经将 STDOUT 重新指定到屏幕，因此这里会把该行内容打印到屏幕上。
13. 将文件 temp 改名为 data，并用 temp 中的内容覆盖写入 data 文件。

输入过滤器。 用户在通过 open 函数创建文件句柄时，还可打开输入过滤器，以便将输入内容转发给 Perl 脚本。此时，命令将以管道符号结尾。

格式：

```
open(FILEHANDLE, COMMAND|);
```

示例 10.24

```
#!/bin/perl
# Scriptname: infilter
```

```
1   open(INPIPE, "date |");     # Windows (2000/NT) use:  date /T
2   $today = <INPIPE> ";
3   print $today;
4   close(INPIPE);

(Output)
Sun Feb 18 14:12:44 PST 2007
```

解释

1. 使用用户定义的文件句柄 INPIPE，把来自输入过滤器的内容重定向为 Perl 程序的输入。本行将通过文件句柄 INPIPE 把 UNIX date 命令的输出内容用作输入源。对于 Windows 2000/NT 用户而言，该命令则应当写作 date /T。
2. 标量 $today 从文件句柄 INPIPE 中接收输入内容。即 Perl 程序从文件句柄 INPIPE 中读取输入。
3. 将 UNIX date 命令的结果赋值给标量 $today，然后打印其值。
4. 在用完文件句柄后，应当使用 close 函数关闭它。这样才能保证在退出脚本之前完成所有命令。如果不关闭文件句柄的话，程序就可能不会正确地清除输出。

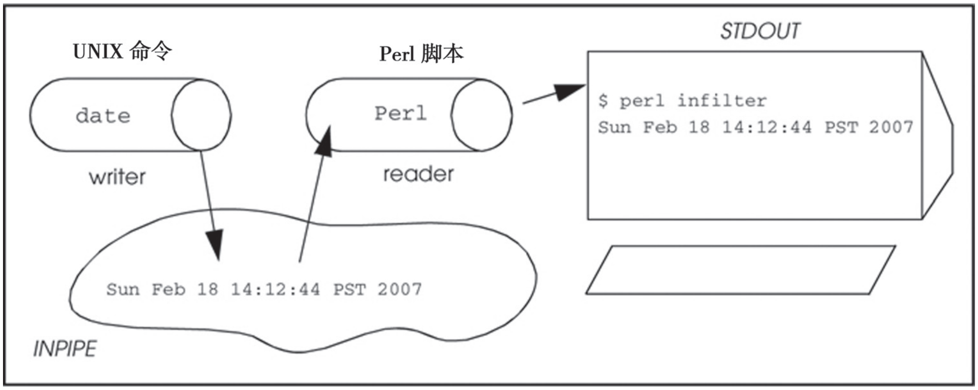


图 10-3　Perl 输入过滤器

示例 10.25

```
(The Script)
1   open(FINDIT, "find . -name 'perl*' -print |") ||
           die "Couldn't execute find!\n";
2   while( $filename = <FINDIT> ){
3      print $filename;
    }

(Output)
3   ./perl2
    ./perl3
    ./perl.man
    ./perl4
    ./perl5
    ./perl6
    ./perl7
    ./perlsub
    ./perl.arg
```

解释

1. 将 UNIX 中 find 命令的输出内容重定向给输入文件句柄 FINDIT。该文件句柄位于角括号中，表明程序的标准输入将来自 FINDIT 文件句柄，而非 STDIN。如果打开文件失败的话，则通过 die 运算符打印信息 Couldn't execute find！，然后退出脚本。
2. 将 UNIX find 命令的输出内容传送给文件句柄 FINDIT。在每一次重复执行 while 循环时，就将来自文件句柄 FINDIT 的一行内容赋值给标量型变量 $filename。
3. print 函数负责将标量 $filename 的值打印到屏幕。

示例 10.26

```
(The Script)
    # Opening an input filter on a Win32 platform
1   open(LISTDIR, 'dir "C:\perl" |') || die;
2   @filelist = <LISTDIR>;
3   foreach  $file ( @filelist ){
        print $file;
   }

(Output)
 Volume in drive C is 010599
 Volume Serial Number is 2237-130A

 Directory of C:\perl

03/31/1999  10:34p      <DIR>          .
03/31/1999  10:34p      <DIR>          ..
03/31/1999  10:37p              30,366 DeIsL1.isu
03/31/1999  10:34p      <DIR>          bin
03/31/1999  10:34p      <DIR>          lib
03/31/1999  10:35p      <DIR>          html
03/31/1999  10:35p      <DIR>          eg
03/31/1999  10:35p      <DIR>          site
               1 File(s)          30,366 bytes
               7 Dir(s)      488,873,984 bytes free
```

解释

1. 将 Windows 中 dir 命令的输出内容重定向给输入文件句柄 LISTDIR。该文件句柄位于角括号中，表明程序的标准输入将来自 dir 文件，而非 STDIN。如果文件打开失败的话，则通过 die 运算符打印错误信息，并退出脚本。
2. 将 Windows 中 dir 命令的输出内容传送给文件句柄 LISTDIR。本行将从文件句柄中读取输入内容，并将它们赋值给数组 @filelist。该数组的每一个元素将各含有一行内容。
3. foreach 循环反复遍历数组内容，每次一行地打印各行内容，直到数组末尾。

10.2 参数传递

10.2.1 ARGV 数组

怎样才能把命令行参数传递给 Perl 脚本呢？如果读者接触过 C、awk 或 C shell 的话，一定会想：“这个我懂！”。不过请注意，Perl 的实现和它们有一些细微的差别。因此，请读者耐心阅读本节下面的内容。

Perl 会把命令行参数保存到特殊数组 ARGV 中，该数组的下标从 0 开始。不过和 C 语言或 awk

命令不同的是，ATGV[0] 在这里并不代表程序名，而是代表着脚本后面的第一个词。与 shell 脚本一样，特殊变量 $0 负责保存 Perl 脚本名。和 C shell 脚本不同，变量 $#ARGV 在 Perl 中负责保存数组最后一个元素的编号，而不是数组元素数目。因此，命令行参数的总数是 $#ARGV+1。$#ARGV 变量的初始值为 −1。

当文件句柄 ARGV 位于角括号中时（即 <ARGV>），Perl 便会把命令行参数当作文件名来对待。此时就会把文件名赋值给 ARGV，并立刻将数组 @ARGV 左移一位，这样也就缩短了 @ARGV 数组的长度。

在上述过程中，Perl 会把移出数组 @ARGV 的元素值赋值给变量 $ARGV。因此，$ARGV 含有当前选择的文件句柄名。

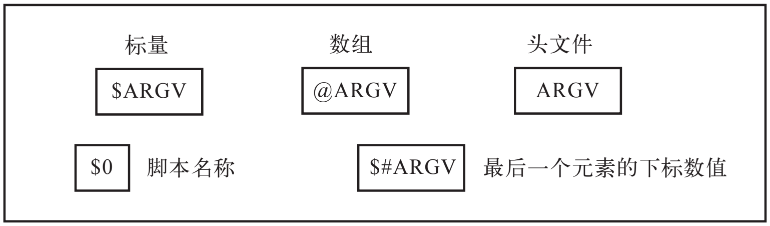


图 10-4　有关 ARGV 的内容

示例 10.27

```
(The Script)
    #!/usr/bin/perl
1   die "$0 requires an argument.\n" if $#ARGV < 0 ;
                            # Must have at least one argument
2      print "@ARGV\n";       # Print all arguments
3      print "$ARGV[0]\n";    # Print first argument
4      print "$ARGV[1]\n";    # Print second argument
5      print "There are ", $#ARGV + 1," arguments.\n";
                            # $#ARGV is the last subscript
6      print "$ARGV[$#ARGV] is the last one.\n";  # Print last arg
(Output)
    $ perl.arg
2   perl.arg requires an argument.

    $ perl.arg f1 f2 f3 f4 f5
2   f1 f2 f3 f4 f5
3   f1
4   f2
5   There are 5 arguments.
6   f5 is the last one.
```

解释

1. 如果没有收到任何命令行参数，则通过 die 函数退出脚本。特殊变量 $0 负责保存 Perl 脚本的名称，即 Perl.arg。
2. 打印数组 @ARGV 的内容。
3. 打印第一个参数值，它不是脚本名。
4. 打印第二个参数值。
5. $#ARGV 变量含有最后一个参数的下标。由于下标是从 0 开始的，因此 $#ARGV+1 表示参数的总数，而非脚本名的长度。
6. 由于 $#ARGV 变量含有最后一个参数的下标，因此 $ARGV[$#ARGV] 是数组 @ARGV 的最后一个元素的值。

10.2.2 ARGV 与 Null 文件句柄

当 ARGV 数组出现在循环表达式中，并位于角括号之间时，其每个元素都会当作是一个特殊的文件句柄。Perl 将会借助数组的移动操作把数组中的每一个元素分别保存到变量 $ARGV 中。一组空的角括号又称作 Null 文件句柄。这时，Perl 会隐式地将 ARGV 数组中的每一个元素当作文件句柄。当使用输入运算符 <> 时，不论是否出现关键字 ARGV，Perl 都会逐一遍历并移动 ARGV 数组中的各个参数，并按顺序依次处理这些参数。在打开 ARGV 文件句柄后，每次只能移出其中一个参数。因此，如果以后还要用到这些参数的话，必须将它们转存到其他数组中去。

示例 10.28

```
(The Text Files)
$ cat f1
Hello there. Nice day.
$ cat f2
Are you sure about that?

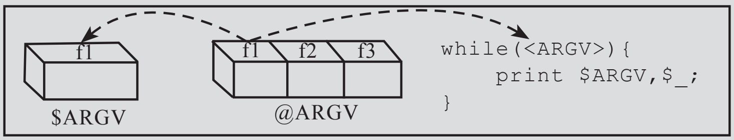
$ cat f3
This is it.
This is the end.

(The  Script)
1   while( <ARGV> ) {print ;}
2   print "The value of \$ARGV[0] is $ARGV[0].\n";

(Output)
$ argv.test f1 f2 f3
Hello there. Nice day.
Are you sure about that?
This is it.
This is the end.
The value of $ARGV[0] is .
```

解释

1. 打印命令行中指定的所有文件内容。执行完毕后，所有参数都将移出数组。本行将依次读取 f1、f2 和 f3 的内容，并打印出来。
2. 由于所有的参数均已移走，因此 $ARGV[0] 不含任何值，本行将不打印任何内容。



示例 10.29

```
(The Text File: emp.names)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
Karen Evich
(The Script)
    # Scriptname: grab.pl
```

```
    # Program will behave like grep -- will search for a pattern
    # in any number of files.
1   if ( ($#ARGV < 1 ) {die "Usage: $0 pattern filename(s) \n";}
2   $pattern = shift;
3   while($line=<ARGV>){
        print "$ARGV: $.: $line" if $line =~ /$pattern/i;
        close(ARGV) if eof;
    }

(Output)
    $ grab.pl
1   Usage: grab.pl pattern filenames(s)
    $ grab.pl 'norma' db
2   db:5: Norma Cord
    $ grab.pl 'Sir Lancelot' db
3   db:4: Sir Lancelot
    $ grab.pl '^.... ' db
4   db:3: Lori Gortz
    $ grab.pl Steve d*
5   datebook.master:12: Johann Erickson:Stevensville, Montana
    datafile:8: Steven Daniels:496-456-5676:83755:11/12/56:20300
    db:1: Steve Blenheim
```

解释

1. 如果没有接收到任何命令行参数，则通过 die 函数退出脚本。
2. 从数组 @ARGV 中移出第一个参数。它含有所要查找的模式。
3. 由于已经把第一个参数从数组 @ARGV 中移出并赋值给了变量 $pattern，因此从命令行得到的其他参数将依次赋值给 ARGV 文件句柄。进入 while 循环，读取一行内容，并将它赋值给 $line 变量。
4. 标量 $ARGV 中含有当前要处理的文件名。变量 $. 中含有当前行号。如果 $pattern 变量值匹配成功的话，就打印找到的文件名、模式匹配的行号、以及该行本身的内容。模式里位于最后一个定界符右侧的 i 字符负责关闭大小写敏感性。
5. 当到达待处理文件的末尾（EOF）时，关闭 ARGV 文件句柄。这样做即可重置变量 $.。如果这里没有显式地关闭 ARGV，变量 $. 就会不断递增下去，并在读取下一个文件时仍无法回到 1。

示例 10.30

```
(The Script)
1   unless ( $#ARGV == 0 ){ die "Usage: $0 <argument>: $!"; }
2   open(PASSWD, "etc/passwd") || die "Can't open: $!";
3   $username=shift(@ARGV);
4   while( $pwline = <PASSWD>){
5       unless ( $pwline =~ /$username:/){ die "$username is not
                                            a user here.\n";}

    }
6   close PASSWD;
7   open(LOGGEDON, "who |" ) || die "Can't open: $!" ;
8   while($logged = <LOGGEDON> ){
       if ( $logged =~ /$username/){ $logged_on = 1; last;}
    }
9   close LOGGEDON;
    die "$username is not logged on.\n" if ! $logged_on;
    print "$username is logged on and running these processes.\n";
10  open(PROC, "ps -aux|" ) || die "Can't open: $! ";
```

```
    while($line=<PROC>){
       print "$line" if  $line =~ /$username:/;
    }
11  close PROC;
    print '*' x 80; "\n";
    print "So long.\n";

(Output)
    $ checkon
1   Usage: checkon <argument>:  at checkon line 6.
    $ checkon joe
5   Joe is not a user here.
    $ checkon ellie
8   ellie is logged on and running these processes:
ellie  3825  6.4  4.5  212  464 p5 R   12:18   0:00 ps -aux
ellie  1383  0.8  8.4  360  876 p4 S   Dec 26 11:34 /usr/local/OW3/bin/xview
ellie  173   0.8 13.4 1932 1392 co S   Dec 20389:19 /usr/local/OW3/bin/xnews
ellie  164   0.0  0.0  100    0 co IW  Dec 20  0:00 -c
             < some of the output was cut to save space >
ellie  3822  0.0  0.0    0    0 p5 Z   Dec 20  0:00 <defunct>
ellie  3823  0.0  1.1   28  112 p5 S   12:18   0:00 sh -c ps -aux | grep '^'
ellie  3821  0.0  5.6  144  580 p5 S   12:18   0:00 /bin/perl checkon ellie
ellie  3824  0.0  1.8   32  192 p5 S   12:18   0:00 grep ^ellie
ellie  3815  0.0  1.9   24  196 p4 S   12:18   0:00 script checkon.tsc
******************************************************************************
```

解释

1. 该脚本只接受一个参数。如果 ARGV 为空（即命令行没有提供任何参数）的话，就执行 die 函数退出脚本，并产生一个出错消息（请记住，$#ARGV 负责保存最后一个参数的编号；@ARGV[0] 是第一个参数，而不是脚本名称；脚本名保存在变量 $0 中）。如果用户提供了多个参数，该脚本在执行时也将产生错误信息。
2. 通过 PASSWD 文件句柄以读方式打开文件 /etc/passwd。
3. 从 @ARGV 中移出第一个参数，并赋值给变量 $username。
4. 每次进入 while 循环时，均通过 PASSWD 文件句柄读取文件 /etc/passwd 的一行内容。
5. 使用 =~ 检查第一个参数内容是否匹配 $username。如果找不到匹配，则退出循环。
6. 关闭文件句柄。
7. 将文件句柄 LOGGEDON 打开为输入过滤器。把 UNIX 中 who 命令的输出内容转发给该文件句柄。
8. 检查输入过滤器的每一行内容。如果用户已登录的话，就把标量 $logged_on 设置为 1，并退出循环。
9. 关闭输入过滤器。
10. 将文件句柄 PROC 打开为输入过滤器。把 UNIX 命令的输出内容转发给该文件句柄。依次通过过滤器读取每一行内容，并将其存入标量 $line。如果 $line 中含有匹配于该用户的项的话，则将该行内容打印到 STDOUT，即终端屏幕。
11. 关闭过滤器。

10.2.3 eof 函数

eof 函数用于检查是否到达文件末尾。如果对文件句柄 FILEHANDLE 的下一次读操作是发生

在文件末尾，或者文件没有打开的话，函数就返回 1。如果没有提供参数，则 eof 函数将返回上一次文件读操作的 eof 状态。带括号的 eof 函数可用在循环体代码内，负责在读取上一个文件句柄时判断其文件末尾状态。如果不带括号的话，该函数则可检查每个已打开文件的末尾状态。

格式：

```
eof(FILEHANDLE)
eof()
eof
```

示例 10.31

```
(The Text File: emp.names)
Steve Blenheim
Betty Boop
Igor Chevsky
Norma Cord
Jon DeLoach
Karen Evich

(In Script)
1   open ( DB, "emp.names") || die "Can't open emp.names: $!";
2   while(<DB>){
3      print if (/Norma/ .. eof);      # .. is the range operator
    }

(Output)
Norma Cord
Jonathan DeLoach
Karen Evitch
```

解释

1. 通过文件句柄 DB 打开文件 emp.names。
2. while 循环每次从文件句柄 DB 中读取一行内容。
3. 当达到含有正则表达式 Norma 的行时，打印该行内容，并打印从 Norma 到文件末尾（eof）的所有内容。

示例 10.32

```
(The Text Files)
$ cat file1
abc
def
ghi

$ cat file2
1234
5678
9101112

(The Script)
    #!/usr/bin/perl
    # eof.p script
1   while(<>){
```

```
2       print "$.\t$_";
3       if (eof){
           print "-" x 30, "\n";
4          close(ARGV);
        }
    }

(Output)
$ eof.p file1 file2
1   abc
2   def
3   ghi
    ---------------------------
1   1234
2   5678
3   9101112
    ---------------------------
```

解释

1. 保存在数组 ARGV 中的第一个参数是 file1。本行在 while 循环表达式中使用了 Null 文件句柄，并以读方式打开文件 file1。
2. 标量 $. 是一个特殊变量，含有当前打开的文件句柄的行号。打印该变量值，随后是一个制表符，最后是该行本身的内容。
3. 如果已到文件末尾位置，则打印一行 30 个短横线。
4. 关闭文件句柄，把 $. 的值重置为 1，以便下一次还能打开文件。在达到文件 file1 的末尾后，脚本将继续处理下一个参数 file2，同样从第 1 行开始。

10.2.4 -i 开关：原位编辑文件

-i 选项的意思是原位编辑文件（editing file in place）。这里的文件均由命令行命名，并保存在数组 @ARGV 中。Perl 会自动把输出文件名改成与输入文件名一样，并将它设定为打印时的默认目标文件。用户如需备份原始文件，则可使用 -i 开关项指定其扩展名，譬如 -i.bak，这样便能把原始文件备份为 filename.bak。若要读取某个文件，则必须先将该文件赋值给文件句柄 ARGV。用户亦可在命令行参数中传入多个文件名，并依次原位编辑每个文件。

示例 10.33

```
(The Text File)
1   $ more names
    igor chevsky
    norma corder
    jennifer cowan
    john deloach
    fred fardbarkle
    lori gortz
    paco gutierrez
    ephram hardy
    james ikeda

(The Script)
2   #!/usr/bin/perl -i.bak
```

```
    # Scriptname: inplace

3   while(<ARGV>){    # Open ARGV for reading
4      tr/a-z/A-Z/;
5      print;    # Output goes to file currently being read in-place
6      close ARGV if eof;
    }

(Output)
7   $ inplace names
    $ more names
    IGOR CHEVSKY
    NORMA CORDER
    JENNIFER COWAN
    JOHN DELOACH
    FRED FARDBARKLE
    LORI GORTZ
    PACO GUTIERREZ
    EPHRAM HARDY
    JAMES IKEDA

8   $ more names.bak
    igor chevsky
    norma corder
    jennifer cowan
    john deloach
    fred fardbarkle
    lori gortz
    paco gutierrez
    ephram hardy
    james ikeda
```

解释

1. 打印源文本文件 names 的内容。
2. 本行表达式中使用了 -i 开关项，因此会原位编辑 names 文件，并将原始文件转存为 names.bak。
3. 进入 while 循环，以读方式打开文件句柄 ARGV。
4. 把待处理文件中所有的小写字母全部转换为大写字母（借助 tr 函数）。
5. print 函数把输出内容转发给正在原位编辑的文件。
6. 当到达文件末尾时，关闭文件句柄 ARGV。这样便能在处理多个文件时及时重置行号，或者在追加文件内容时标记文件的末尾位置。
7. 由输出结果可见，程序更改了 names 文件内容。这说明原位修改操作成功。
8. 创建文件 names.bak，并将其作为原始文件的备份，因为原始文件已经更改了。

10.3 文件测试

与 shell 一样，Perl 也提供了许多文件测试运算符（详见表 10.3），用于查看各种文件属性，如文件是否存在、访问权限、是否目录、是否文件等。其中大部分的运算符都在结果为真时返回 1，在结果为假时返回“”（即 null）。

如果需要反复多次测试同一个文件，则可在程序中使用单个下划线来代表文件名。这样便可利用前一次文件测试的 stat 结构。

表 10-3 文件测试运算符 [a]

运算符	含 义
-r $file	如果 $file 可读，则为真
-w $file	如果 $file 可写，则为真
-x $file	如果 $file 可执行，则为真
-o $file	如果 $file 的属主是有效的 uid，则为真
-e $file	如果 $file 存在，则为真
-z $file	如果 $file 大小为 0，则为真
-s $file	如果 $file 大小非 0，则为真。返回文件字节大小
-f $file	如果 $file 是普通文件，则为真
-d $file	如果 $file 是目录，则为真
-l $file	如果 $file 是符号链接，则为真
-p $file	如果 $file 是命名的管道或 FIFO，则为真
-S $file	如果 $file 是套接字，则为真
-b $file	如果 $file 是块特殊文件，则为真
-c $file	如果 $file 是字符特殊文件，则为真
-u $file	如果 $file 具有 setuid 位设置，则为真
-g $file	如果 $file 具有 setgid 位设置，则为真
-k $file	如果 $file 具有 sticky 位设置，则为真
-t $file	如果 $file 文件句柄对 tty 打开，则为真
-T $file	如果 $file 是文本文件，则为真
-B $file	如果 $file 是二进制文件，则为真
-M $file	返回上一次修改文件后经过的天数
-A $file	返回上一次访问文件后经过的天数
-C $file	返回信息结点改变后经过的天数

a. 如果没有提供文件名的话，则设置默认文件名为 $_。

示例 10.34

```
(At the Command Line)
1   $ ls -l perl.test
    -rwxr-xr-x  1 ellie          417 Apr 23 13:40 perl.test
2   $ ls -l afile
    -rws--x--x  1 ellie            0 Apr 23 14:07 afile

(In Script)
    #!/usr/bin/perl
    $file=perl.test;
```

```
3   print "File is readable\n" if -r  $file;
    print "File is writeable\n" if -w  $file;
    print "File is executable\n" if -x  $file;
    print "File is a regular file\n" if -f  $file;
    print "File is a directory\n" if -d $file;
    print "File is text file\n" if -T $file;
    printf "File was last modified %f days ago.\n", -M $file;
    print "File has been accessed in the last 12 hours.\n" if -M <= 12;
4   print "File has read, write, and execute set.\n"
           if -r $file && -w _ && -x _;
5   stat("afile");  # stat another file
    print "File is a set user id program.\n" if  -u _;
                    # underscore evaluates to last file stat'ed
    print "File is zero size.\n" if -z_;

(Output)
3   File is readable
    File is writeable
    File is executable
    File is a regular file
    *** No print out here because the file is not a directory ***
    File is text file
    File was last modified 0.000035 days ago.
    File has read, write, and execute set.
    File is a set user id program.
    File is zero size.
```

解释

1. 打印文件 Perl.test 的权限、属主、文件大小等信息。
2. 打印文件 afile 的权限、属主、文件大小等信息。
3. 如果文件是可读可写且可执行的话，执行 print 语句。
4. 由于这里需要检查同一个文件的多个属性，因此要给文件测试运算符加上下划线。下划线表明将引用已有的 stat[a] 结构，后者是一个含有文件信息的数组。
5. stat 函数返回一个 13 元数组，其中含有文件相关的统计信息。如果为文件测试运算符加上下划线的话，就会在随后的测试中始终使用 afile 文件的统计信息。

a. 有关 stat 结构的更多信息请参阅第 18 章“与系统交互”。

10.4 读者应当学到的知识

1. 什么是文件句柄？
2. “以读方式打开文件”是什么意思？
3. 在以写方式打开文件时，如果该文件已存在的话，将会发生什么情况？
4. select() 函数的用途是什么？
5. 什么是 binmode ？
6. 在处理文件时，die 函数的用途是什么？
7. Windows 与 UNIX 系统处理行终止符的方式有什么不同？
8. 什么是排他锁（exclusive lock）？
9. tell() 函数的返回值是什么？

10. +< 标记与 +> 标记有何区别？
11. stat() 函数的用途是什么？
12. 怎样才能在一个文件中重定位其文件指针？
13. 在测试文件属性时，-M 开关项的功能是什么？

10.5 下章简介

迄今为止，本书介绍的所有函数都是 Perl 直接提供的。它们都是所谓的内建函数（build-in function），譬如 print()、printf()、push()、pop()、chomp() 等。在使用这些函数时，用户只需知道它们的用途和调用格式，而不必关心 Perl 开发者是如何让它们工作的。在下一章里，读者将学习如何编写属于自己的函数，又称子例程（subroutine）；并学习如何向它们发送消息，以及如何返回其结果。

练习 10　获得句柄

练习 A

1. 创建文件句柄，以读方式打开文件 datebook（可从华章网站 (www.hzbook.com) 上找到该文件），并把收入大于 $50,000 的人名打印到另一个文件句柄。
2. 要求用户为文件 datebook 输入新数据（包括姓名、电话号码、地址等，保存在单独的标量中）。然后通过用户定义的文件句柄向 datebook 文件追加换行符。

练习 B

1. 借助过滤器，按照姓氏顺序排列文件 datebook。
2. 使用 open 函数创建文件句柄，通过输入过滤器列出当前目录下的所有文件，并打印所有可读的文本文件内容。如果 open 失败，则通过 die 函数退出脚本。
3. 重写程序，检查列出的文件在过去 12 个小时是否经过修改，并打印这些文件的名字。

练习 C

1. 在文件 datebook 中创建多个重复的项。例如，Fred Fardbarkle 重复出现 5 次，Igor Chevsky 重复出现 3 次，等等。在大多数编辑器中，这些只需简单的复制 / 粘贴操作。

   编写程序，将文件 datebook 的文件名赋值给标量，并检查文件是否存在。如果存在的话，程序继续检查文件是否可写或可读，并通过 die 函数发送错误信息到屏幕。同时还要告诉用户 datebook 文件上一次修改的时刻。

   程序还应读取 datebook 文件的每一行内容，并将每个人的薪水增加 10%。不过，如果有人在文件中不止出现一次（假定同名同姓就代表重复）的话，则只处理第一次，而跳过第二次出现的情况。程序应将每一行输出发送到文件 raise 中。在 raise 文件内，任何人都不应该重复出现，并且其薪水值都应当增加 10%。

   在屏幕上显示 datebook 文件中所有人的平均收入。对于重复的项，应当打印重复的人名以及相应的重复次数。
2. 编写脚本 checking，以任意数量文件作为命令行参数，打印可读并且可写的文本文件名字。如果用户没有提供任何参数，则打印错误信息并退出脚本。

第 11 章　子例程与函数

11.1　子例程 / 函数

除了 Perl 已经提供的大量函数之外，用户还可以创建自己的函数或子例程。有些语言是区分函数（function）与子例程（subroutine）这两个词的，但 Perl 不会。如果说“子例程”，大家都知道这指的是“函数”；而如果提到“函数”，则每个人都清楚它指的是“子例程”[⊖]。从技术层面上讲，函数是一段能提供返回值的代码，而子例程则是执行某些任务的代码块，后者不返回任何结果。Perl 的子例程与函数都能同时支持上述两种模式。因此，本章将互换地使用这两个术语。本书将在涉及用户自定义函数时使用“子例程”一词。

子例程是一组自包含的程序单元，其设计目标是完成指定的任务，譬如计算抵押贷款额度、从数据库中检索数据、或者检查输入内容的合法性。当在程序中调用子例程时，就好像离开主程序走另一条捷径一样。Perl 接着会开始执行子例程中的指令，并在执行完毕后返回主程序中原先的调用位置继续运行。子例程可以反复执行，因此它能帮助开发者避免重复编码。子例程还可用于把一个大程序拆分为更小的模块，从而提升代码的可组织性和可维护性。

子例程声明（declaration）是由一个或多个语句构成代码块，它独立于主程序，并且只有在调用到时才能执行。一般而言，读者可以将子例程看成是一个“黑盒子”。输入信息进入这个黑盒子（譬如在计算器或遥控器上按下按钮），然后由黑盒子返回相应的输出信息（譬如计算结果或相应电视频道）。在这个过程中，黑盒子内部的处理流程是对用户完全透明的。惟一关心其内部细节的人就是编写这个子例程的程序员。当使用 Perl 内建函数（如 print() 或 rand() 函数）时，只需把一个文本字符串或一个数字传入该函数，然后便可从函数得到结果。用户无需关心它是怎么做到这些的，只需要相信这些函数将不负所托地完成任务。如果送入的数据信息有误，其得到的结果也就自然是不正确的，一言以蔽之：“传入垃圾，得到的也将是垃圾”。

子例程的作用域（scope）是指在程序中能够看到它的位置范围。子例程是全局的，可以放在脚本中的任意位置，甚至放在其他脚本文件中。当使用来自其他文件的子例程时，应当使用关键字 do、require 或 use 将它加载到脚本里。在子例程中创建或访问的所有变量也是全局的，除非用户专门通过 local 或 my 运算符将它声明为局部变量。

若要调用一个子例程，用户可以在子例程名之前加上 & 符号，或在子例程前面加上 do 运算

⊖　您甚至可以把本章标题改成“函数与子程序”。不过这样的话有些人就会不赞成啦。

符[⊖]，也可在子例程名后面加上一组括号。如果使用了向前引用（forward reference）的话，在调用子例程时就不需要提供 & 或括号。

如果调用的子例程不存在，程序便会退出执行，并产生错误消息：Undefined subroutine “main：：prog”……如要检查子例程是否已有定义的话，可使用内建的 defined 函数。

子例程的返回值是其最后一个求值表达式的值（标量型或数组型）。用户亦可通过 return 函数显式地返回它的结果值，或者根据某些条件的测试结果，然后直接从子例程中退出来。

如果在表达式内部调用子例程的话，可直接将子例程返回值赋予指定变量。也就是说和函数的使用方式一模一样。

格式

```
Subroutine declaration:
    sub subroutine_name;
Subroutine definition:
    sub subroutine_name { Block }
Subroutine call:
    do subroutine_name;
    &subroutine_name;
    subroutine_name();
    subroutine_name;
Subroutine call with parameters:
    &subroutine_name(parameter1, parameter2, ... )
    subroutine_name(parameter1, parameter2, ... )
```

定义并调用子例程

声明（declaration）只是向 Perl 编译器说明要在程序中定义一个子例程，并规定它的参数格式。子例程声明的作用域是全局的，换而言之，不论将它放在程序中的哪个位置，它们都将是可见的。不过习惯上一般会把子例程的声明语句放在程序的开头或末尾。所谓子例程的定义（definition），则是跟随在子例程名后面的代码块。如果尚未显式地声明该子例程，则会在定义该子例程时一并声明它。

声明：　　sub 子例程名；

定义：　　sub 子例程名 { 语句 ; 语句 ; }

子例程的定义可以出现在程序中的任何位置，甚至在其他文件中。一个子例程包含一个关键字 sub，其后是一个起始花括号、位于花括号之间的一组语句、以及一个闭合花括号。

在调用之前，子例程是不会自己执行的。若要调用一个子例程，可在子例程名之前加上一个 & 符号，也可在子例程名后面加上一组空的括号，还可通过上述方式以内建函数的形式调用它。如果在调用子例程时既不提供 & 符号又不加小括号，则必须在调用前事先声明该子例程。

示例 11.1

```
(The Script)
sub greetme { print "Welcome, Välkommen till, Bienvenue!\n";}
&greetme if defined &greetme;
print "Program continues....\n";
```

[⊖] do 函数的主要用途是从 Perl4 的库（譬如 do’ pwd.pl）中载入 Perl 子程序。

```
4   &greetme; # Call to subroutine
5   print "More program here.\n";
6   &bye;
7   sub bye { print "Bye, adjo, adieu.\n"; }
8   &bye;

(Output)
2   Welcome, Välkommen till, Bienvenue!
3   Program continues....
4   Welcome, Välkommen till, Bienvenue!
5   More program here.
    Bye, adjo, adieu.
    Bye, adjo, adieu.
```

解释

1. 这是一个子例程的定义。开头是关键字 sub，其后跟随着子例程名 greetme，最后则是调用时需要执行的代码块。一般可以在子例程名前面缀以 & 符号，不过这不是必需的。惟一必需加 & 符号的情况出现在调用子例程时、通过子例程名创建引用时、以及为函数（如 defined）提供参数时。该定义可以放在程序的任何位置，并且在调用前不会对程序产生任何影响。本例在调用函数时将只执行一条 print 语句。
2. 通过在子例程名前面加上 & 标记的方式调用子例程 greetme。这里使用内建函数 defined 检查该子例程是否已经定义。在使用子例程名时，要求必须提供 & 标记。在调用时，程序将跳入子例程中，并执行后者定义的语句，在这里即为 Welcome 语句。
3. 当子例程调用完毕后，程序将从原位置的下一条语句开始继续执行主程序。
4. 再一次调用子例程 greetme。
5. 子例程在第 4 行退出，然后恢复执行原程序。
6. 调用子例程 bye。在后面第 7 行中可以找到该子例程的定义。
7. 定义子例程 bye。不论把子例程定义放在哪里，编译器都能找到它。
8. 调用子例程 bye。

空值（null）参数列表。如果子例程名后面跟随着空括号（即空值参数列表），则可以不在其程序名前面加 & 标记。

示例 11.2

```
#!/usr/bin/perl
$name="Ellie";
print "Hello $name.\n";

bye();    # Without parens or an ampersand, bye would be a bareword
          # causing a warning message when -w is used.
sub bye{
    print "Bye $name.\n";
}
```

向前引用。向前引用（forward reference）负责向编译器声明已经在程序中的某个位置定义好了子例程。如果出现了向前引用，则在调用该子例程时就不必提供 & 标记。

示例 11.3

```
    #!/usr/bin/perl
1   sub bye;  # Forward reference

    $name="Ellie";
2   print "Hello $name.\n";

3   bye;      # Call subroutine without the ampersand

4   sub bye{
5       print "Bye $name\n";
    }
(Output)
2   Hello Ellie.
5   Bye Ellie
```

变量作用域。作用域（scope）负责描述变量在程序中的哪些位置是可见的。Perl 变量的作用域都是全局的，即在整个程序中都是可见的，就连子例程中的变量也不例外。如果在子例程中声明了一个变量，则该变量将对整个程序可见。如果在子例程中修改已有变量值，则在退出子例程后这次修改仍旧有效。局部（local）变量是只能用于局部代码块、子例程或文件的变量。用户必须使用内建函数 local 或 my 来创建局部变量，因为在默认情况下 Perl 变量的作用域都是全局的（详见“使用 local 和 my 按值调用”一节）。

示例 11.4

```
(The Script)
    # Script: perlsub_sub2
    # Variables used in subroutines are global by default
1   sub bye { print "Bye $name\n"; $name="Tom";}
                                # Subroutine definition
2   $name="Ellie";
3   print "Hello to you and yours!\n";
4   &bye;
5   print "Out of the subroutine. Hello $name.\n";
                                # $name is now Tom
6   &bye;

(Output)
3   Hello to you and yours!
1   Bye Ellie
5   Out of the subroutine. Hello Tom.
1   Bye Tom
```

解释

1. 定义子例程 bye。在子例程块中将变量 $name 赋值为 Tom。$name 是一个全局变量，也就是说它在整个程序中都是可见的 [a]。
2. 从本行开始执行程序。将全局变量 $name 赋值为 Ellie。
3. 本行只负责说明执行的流程。
4. 调用子例程 &bye。程序在第一行处进入子例程，此时 $name 的值仍然是 Ellie。在执行完 Bye 行之后，会给变量 $name 赋予新值 Tom。然后退出子例程，从第 5 行恢复程序执行。

5. 在子例程中改变变量 $name 的值。
6. 再次调用子例程。此时，变量 $name 的值为 Tom。

a. 这里假设把程序编译到名叫 main 的包中。有关包和作用域的内容请参阅第 12 章“模块、包和库”。

11.2 参数传递

在调用子例程时，一般需要在括号中提供一组以逗号隔开的参数。

```
The feed_me function below takes 3 arguments when called:

@fruit=qw(apples pears peaches plums);  # Declare variables
$veggie="corn";

&feed_me( @fruit, $veggie, "milk" );  # Call subroutine with arguments
```

参数可以由数字、字符串、引用、变量等组成。函数都把参数保存到特殊的 Perl 数组中去，该数组名叫 @_，其中每个元素称作是一个参数。

```
sub feed_me{ print join(",", @_),"\n"; }
# Subroutine gets arguments in @_ array

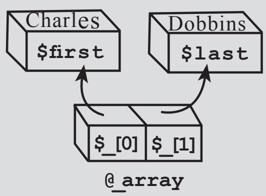
Output:  apples, pears, peache, plums, corn, milk
```

按引用调用和 @_ 数组。 不论参数是标量型还是数组型的，都必须将它们传入子例程，并保存到 @_ 数组中去。数组 @_ 是一个局部（local）数组，其值是隐含的对实际参数的引用。如果修改 @_ 数组，就必须同时修改这些实际的参数。如果只是移动或弹出 @_ 数组中的元素，则只会失去对实际参数的引用而已（详见“使用 local 和 my 按值调用”一节）。

当把数组或标量传递给函数或者子例程时，Perl 默认通过按引用调用（call by reference）的方式调用它们。@_ 数组是一个特殊的局部数组，用于引用实际参数的名字。用户可以改变 @_ 数组的值，因此便可更改相应实际参数的值。@_ 数组的元素包括 $_[0]、$_[1]、$_[2] 等。如果传递的是标量型参数，其参数值就是 @_ 数组的第一个元素 $_[0]。Perl 并不关心这些传进来的参数是否全被用到，或者是否有足够的参数传递进来。如果对 @_ 数组执行移动或退出操作，则只会失去对实际参数的引用。此外，如果要更改全局拷贝而不是局部数组 @_ 的话，用户可以使用 typeglobs（符号引用）或者指针（硬引用）。本书将在下面“通过指针传递”一节以及第 13 章的“这个作业需要引用吗？”一节予以详细讨论。

示例 11.5

```
(The Script)
    #Passing arguments
1   $first="Charles";
    $last="Dobbins";
2   &greeting ( $first, $last );
3   sub greeting{
4      print "@_", "\n";
5      print "Welcome to the club, $_[0] $_[1]!\n";
6   }
```



```
(Output)
4   Charles Dobbins
5   Welcome to the club, Charles Dobbins!
```

解释

1. 给标量赋值。
2. 使用两个参数 $first 和 $last 调用子例程 greeting。
3. 声明子例程。
4. 参数保存在数组 @_ 中，它是一个局部数组，在进入子例程时自动创建，并在退出子例程时自动销毁。数组中含有对两个参数 $first 和 $last 的引用。
5. 打印数组 @_ 中的前两个元素。标量 $_[0] 和 $_[1] 分别代表单个元素。
6. 闭合花括号说明子例程在这里结束。数组 @_ 自动销毁。

示例 11.6

```
(The Script)
   # Program to demonstrate how @_references values.
1  sub params {
2     print 'The values in the @_array are ',"@_\n";
3     print "The first value is $_[0]\n";
4     print "The last value is ",pop(@_),"\n";
5     foreach $value ( @_ ) {
6         $value+=5
          print "The value is $value", "\n";
      }
   }

   print "Give me 5 numbers : ";
7  @n=split(' ',<STDIN>);
8  &params(@n);
   print "Back in main\n";
9  print "The new values are @n \n";

(Output)
   Give me 5 numbers: 1 2 3 4 5
2  The values in the @_array are 1 2 3 4 5
3  The first value is 1
4  The last value is 5
   The value is 6
   The value is 7
   The value is 8
   The value is 9
9  Back in main
10  The new values are 6 7 8 9 5
```

解释

1. 定义子例程 params。
2. 打印 @_ 的值，即实际参数列表。@_ 是一个局部数组，负责引用传给子例程的所有参数。
3. 打印 @_ 数组的第一个元素。
4. 使用 pop 函数删除数组的最后一个元素，并打印。
5. foreach 循环依次把 @_ 数组中的每个元素赋值给标量 $value。
6. 将数组中每个元素递增 5，并保存到标量 $value 中。

7. 当用户输入 5 个数字后，split 函数返回一个数组，其元素是从 STDIN 文件句柄读取的每一个字符。
8. 以数组为参数调用子例程。
9. 打印出来的值说明在函数中修改的变量值真的改变了。只有原数组最后一个元素的值没有发生变化，因为它在程序第 4 行已被弹出了。

使用 local 和 my 按值调用。大多数编程语言都支持把参数值传递给子例程的方式，该方式将不会改变原参数的值。当以按值调用（call-by-value）方式传递参数时，程序会把参数值的一个副本发送给子例程。即使程序修改了这个副本的值，原参数的值仍将保持不变。在 Perl 中为参数创建副本时，程序会将该参数从 @_ 数组中拷贝出来，然后赋值给另一个局部变量，参见图 11-1。为支持局部副本的创建，Perl 提供了两种内建函数：local 和 my。

local 函数

在 Perl5 出现之前，local 函数负责在 Perl 脚本中打开按值调用（call by value）模式。后来，Perl5 引入了另一种函数 my，能进一步保障变量在代码块中的私有性。

local 函数能从参数列表中创建局部变量。任何使用 local 函数声明的变量都是动态拷贝的。也就是说，该变量不但在创建它的块中可见，而且对于任何从该代码块调用的函数或在嵌入定义的代码块中也都是可见的。如果局部变量和全局变量重名的话，程序将临时创建一个新的局部变量来暂存全局变量的值。当局部变量超出其作用域时，全局变量将再一次可见，并恢复其原有值。在执行完子例程的最后一条语句之后，程序将清除其创建的局部变量。建议读者在子例程代码块的开头设置所有的局部变量。

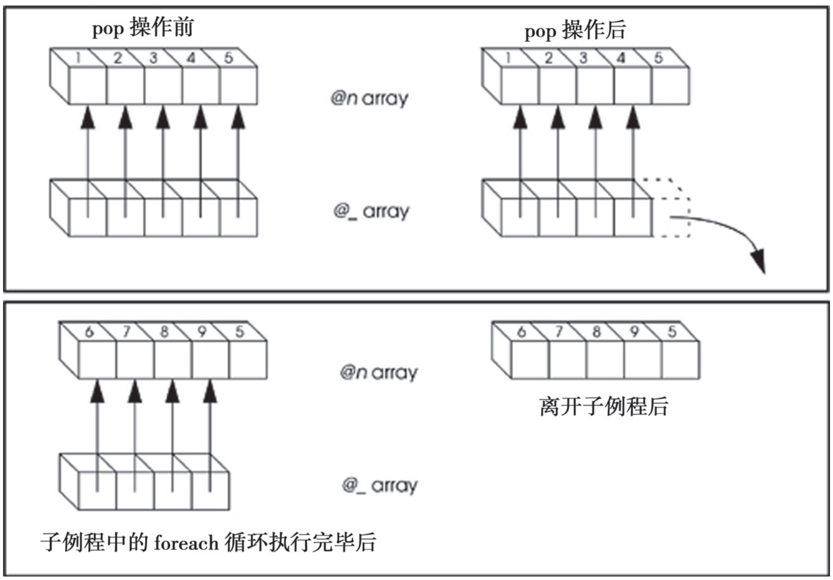


图 11-1 @_ 数组引用实际的标量变量

示例 11.7

```
(The Script)
$first="Per";
```

```
    $last="Lindberg";
2   &greeting ( $first, $last ) ;    # Call the greeting subroutine
3   print "---$fname---\n" if defined $fname;   # $fname is local to
                                                # sub greeting

    # Subroutine defined
    sub greeting{
4       local ($fname, $lname) = @_ ;  # Call by value
5       print "Welcome $fname!!\n";
    }
(Output)
3   <no output>
5   Welcome Per!!
```

解释

1. 为标量型变量赋值。
2. 调用子例程 greeting，并向它传递两个参数。
3. 不执行 print 函数，因为这里没有定义 $fname。它在子例程中定义为一个局部变量，只有对子例程 greeting 而言才是局部（local）的。
4. local 函数从 @_ 数组中获取参数列表，并从该列表创建两个局部变量，$fname 和 $lname。局部变量中的值是传递进来的参数副本。
5. 执行 print 语句，打印局部变量 $fname 的内容。其内容是变量 $first 值的副本。

my 运算符

my 运算符用于在词法上（lexically）打开按值调用（call-by-value）模式。my 声明得到的变量在从声明之处开始到最内层包含的块中都是可见的。这里所谓的块（block）包括所有位于花括号中的代码块、子例程或文件。使用 my 运算符声明的变量将创建在特殊缓冲区中，该缓冲区将专用于创建它的代码块[⊖]。

与 local 函数声明的变量不同，my 函数声明得到的所有变量都只在声明它的子例程中可见，对该子例程调用的任何其他子例程都是不可见的。如果列出多个变量，则必须将它们放在括号中。因此，示例 11.7 中的子例程可按如下方式提供：

```
sub greeting{
    my ($fname, $lname) = @_;  # $fname and $lname are private
    print "Welcome $fname!!\n";
}
```

示例 11.8

```
    (The Script)
    #  The scope of my variables
my $name = "Raimo";
print "$name\n";
    {  # Enter block
       print "My name is $name\n";
       my $name = "Elizabeth";
       print "Now name is $name\n";
       my $love = "Christian";
       print "My love is $love.\n";
```

⊖ 有关 my 变量的更多知识，请参见第 12 章。

```
9        }  # Exit block
10  print "$name is back.\n";
11  print "I can't see my love,$love, out here.\n";

(Output)
2   Raimo
5   My name is Raimo
6   Now name is Elizabeth
8   My love is Christian.
10  Raimo is back.
11  I can't see my love,, out here.
```

解释

1. 使用 my 函数在词法上创建变量 $name，并将它赋值为 Raimo。该变量在创建它的位置以及所有内部代码块中都是可见的。它位于自己专有的缓冲区中。
2. 打印该变量值。
3. 进入新代码块。
4. 词法变量 $name 仍在作用域中，即仍然可见。
5. 声明一个新变量。该变量获得一个专用的缓冲区。
6. 新变量 $name 是可见的，打印其变量值。
7. 在代码块中声明另一个词法变量，并为其提供专用的缓冲区。
8. 打印 $love 的值 Christina。它在本代码块中是可见的。
9. 代码块到此结束。my 变量离开其作用域。
10. 变量 $name 现在是可见的了。打印 Raimo。
11. $love 变量超出了作用域。

示例 11.9

```
(The Script)
    # Difference between my and local
1   $friend="Louise";         # Global variables
2   $pal="Danny";
3   print "$friend and $pal are global.\n";
4   sub guests {
5       my $friend="Pat"; # Lexically scoped variable
6       local $pal="Chris";  # Dynamically scoped variable
7       print "$friend and $pal are welcome guests.\n";
8       &who_is_it;           # Call subroutine
    }

9   sub who_is_it {
10      print "You still have your global friend, $friend, here.\n";
11      print "But your pal is now $pal.\n";  # Dynamically scoped
    }

12  &guests;                  # Call subroutine
13  print "Global friends are back: $friend and $pal.\n";

(Output)
3   Louise and Danny are global.
7   Pat and Chris are welcome guests.
10  You still have your global friend, Louise, here.
```

```
11  But your pal is now Chris.
12  Global friends are back: Louise and Danny.
```

解释

1. 将变量 $friend 赋值为 Louise。在 main 中的所有变量都是全局的。
2. 将变量 $pal 赋值为 Danny，它也是全局的。
3. 打印全局变量值。
4. 定义子例程 guests。
5. 通过 my 函数将变量 $friend 声明为子例程局部变量。
6. 通过 local 函数将变量 $pal 声明为该子例程及其调用的其他子例程的局部变量。
7. 在 guests 子例程中打印 $friend 和 $pal 的变量值。
8. 调用子例程 who_is_it。
9. 定义子例程 who_is_it。
10. 在子例程 who_is_it 中，全局变量 $friend（值为 Louise）是可见的。词法变量 $friend（值为 Pat）在该子例程中是不可见的，因为它是 my 函数在其他子例程中声明的。
11. 另一方面，在子例程 guests 中通过 local 函数声明标量 $pal（值为 Chris），该变量在子例程 who_is_it 中仍然是可见的。
12. 调用 guests 子例程。
13. 退出子例程后，全局变量返回其作用域，并打印它们的值。

使用 strict 编译指示符（my 或 our）。 编译指示符（pragma）是一种程序模块，负责在探测到程序有问题时去触发编译器的相关错误信息。strict 编译指示符可用于在程序中避免使用全局变量。一旦声明了 strict，只要使用了全局变量，即使用 local 函数声明它，编译器也会提示报错。不过词法变量是允许出现的。这些变量都通过内建函数 my 或 our 声明。如果用户需要使用全局变量，同时又想启动 strict 编译指示符以避免在程序其他位置误用全局变量的话，可使用内建函数 our（Perl 5.6 +）。（有关 strict 和包的更多信息，请参阅“strict 编译指示符”一节。）

示例 11.10

```
(The Script)
use strict "vars";
my $name = "Ellie";             # my (lexical) variables are okay
@friends = qw(Tom Stefan Bin Marie);
                                # global variables not allowed
local $newspaper = "The Globe"; # local variables are not allowed
print "My name is $name and our friends are @friends.\n";

(Output)
3   Global symbol "@friends" requires explicit package name at
        rigid.pl line 3.
4   Global symbol "$newspaper" requires explicit package name at
        rigid.pl line 4.
    In string, @friends now must be written as \@friends at rigid.pl
        line 5, near "$name and our friends our @friends"
    Global symbol "@friends" requires explicit package name at
        rigid.pl line 5.
    Execution of rigid.pl aborted due to compilation errors.
```

解释

1. 以 vars 为参数使用 strict 编译指示符，告诉编译器如果发现任何全局变量就提示报错。strict 模块 strict.pm 是标准 Perl 发布包的一部分。
2. 变量 $name 是通过 my 函数定义的词法变量，也就是说，它专属于创建它的代码块。strict 编译指示符与 my 变量类似。
3. 数组 @friends 是一个全局变量。编译器如果碰到像第 3 行输出所示的全局变量，就会提示出错。该消息通过 explicit package name 表明，如果在名称前面加上包名和双冒号（即 @main::friend）的话，就仍然可以使用原来的全局变量。
4. 在动态分配全局变量时，Perl 通过 local 函数来对声明的变量进行分类。由于没有使用 my 函数声明变量，因此在这里编译器仍然会提示报错。如要使用局部变量，则显然应当使用 local $main::newspaper。
5. 由于编译器出错，因此本程序将永远不会执行到这一行。

示例 11.11

```
(The Script)
1   use strict "vars";
2   my $name = "Ellie";        # All variables are lexical in scope
3   our @friends = qw(Tom Stefan Bin Marie);
4   our $newspaper = "The Globe";
5   print "$name and $friends[0] read the $newspaper.\n";

(Output)
5   Ellie and Tom read the The Globe.
```

解释

1. 以 vars 为参数使用 strict 编译指示符，告诉编译器如果发现任何全局变量就提示报错。strict 模块 strict.pm 是标准 Perl 发布包的一部分。
2. 变量 $name 是通过 my 函数定义的词法变量，也就是说，它专属于创建它的代码块。strict 编译指示符与 my 变量类似。
3. our 变量（Perl 5.6+）伪装成一个词法变量 [a]，因此 strict 编译指示符将忽略它的存在。这样便可在真正需要时避免使用全局变量。
4. 这个标量也是 our 变量。它是全局的，也是一个词法变量，因此能避免引发 strict 编译指示符报错。
5. print 函数负责打印词法变量的值。

a. Wall, L., Christianson, T., and Orwant, J., *Programming Perl 3rd ed*., O' Reilly & Associates; Sebastopol, CA , 2000, p. 138。

11.2.1 原型

原型（Prototype）又称作模板（Template），它负责在调用子例程时告诉编译器应当读取多少个参数，以及相应参数类型。它能让 Perl 像对待内建函数一样对待子例程。作为声明的一部分，原型将在程序编译阶段进行处理。如要调用使用原型声明的子例程，则必须省略 & 符号，否则程序便会将该子例程当作用户自定义的子例程来处理，而不是作为内建子例程处理，同时编译器将忽略子例程原型。

```
Prototype: (Perl 5.003+)
    sub subroutine_name($$);
    Takes two scalar arguments

    sub subroutine_name(\@);
    Argument must be an array, preceded with an @ symbol

    sub subroutine_name($$;@)
    Takes two scalar arguments and an optional array.
    Anything after the semi-colon is optional.
```

示例 11.12

```
    # Filename: prototypes
    # Testing prototyping
1   my $a=5;
    my $b=6;
    my $c=7;
2   @list=(100,200,300);
3   sub myadd($$) {        # myadd requires two scalar arguments
        my($x, $y)=@_;
        print $x + $y,"\n";
    }
4   myadd($a, $b);         # Okay
5   myadd(5, 4);           # Okay
6   myadd($a, $b, $c);     # Too many arguments

(Output)
6   Too many arguments for main::myadd at prototypes line 14,
        near "$c)" Execution of prototypes aborted due to compilation
        errors.11
```

解释

1. 声明三个标量型变量，并为它们赋值。
2. 为数组 @list 赋值。
3. 定义子例程 myadd 的原型，期望两个标量型参数。多于或少于两个参数都会造成编译出错。
4. 向子例程提供两个标量型参数。这样就没有问题。
5. 向子例程传递两个数字。这样也没有问题。
6. 子例程原型期望两个标量型参数，但这里要传递三个参数，因此编译器会发送错误消息。

示例 11.13

```
    # Prototypes
1   sub mynumbs(@$;$);        # Declaration with prototype
2   @list=(1,2,3);
3   mynumbs(@list, 25);
4   sub mynumbs(@$;$) {       # Match the prototypes
5       my ($scalar)=pop(@_);
6       my(@arr) = @_;
7       print "The array is: @arr","\n";
8       print "The scalar is $scalar\n";
    }

(Output)
7   The array is: 1 2 3
```

```
8   The scalar is: 25
```

解释

1. 这里是一个原型声明，期望输入数组、变量或者可选的标量。这里的分号表明参数是可选的。
2. 为数组 @list 赋值。
3. 以一个列表和标量值 25 为参数调用子例程 mynumbs。在调用子例程原型时，无需使用 & 符号。
4. 定义子例程。即使已经在子例程声明的第一行中创建过原型，也必须在这里再重复一遍。否则会得到如下错误消息：

   `Prototype mismatch: sub main::mynumbs(@$;$) vs none at prototype line 19。`
5. 弹出 @_ 数组中的最后一个元素，并赋值给标量 $scalar。
6. 将数组 @_ 的剩余元素赋值给 @arr。
7. 打印 @arr 数组的内容。
8. 打印标量 $scalar 的值。

11.2.2 返回值

在调用时，子例程可以表现得类似于函数，即可以返回一个标量或列表。例如，用户可以在赋值运算符的右侧调用子例程。然后，子例程就会把返回值传递给运算符左侧的标量型或数组型变量。

$average = &ave(3,5,6,20);

返回值　　调用子例程

子例程的返回值是其中最后一个表达式的值。

此外，用户还可通过 return 函数返回指定的值，或者根据某些条件提前从子例程中返回。如果在子例程外使用，return 函数将会导致致命错误。可以这样讲：return 是专门用于子例程的；而 exit 则是专用于程序主体的。如果在子例程中使用 exit 函数，则会退出整个程序脚本，并返回命令行。

示例 11.14

```
(The Script)
    #!/bin/perl
    sub MAX {
1       my($max) = shift(@_);
2       foreach $foo ( @_ ){
3           $max = $foo if $max < $foo;
            print $max,"\n";
        }
        print "------------------------------\n";
4       $max;
    }
    sub MIN {
        my($min) = pop( @_ );
        foreach $foo ( @_ ) {
            $min = $foo if $min > $foo;
            print $min,"\n";
        }
        print "------------------------------\n";
        return $min;
    }

5   my $biggest = &MAX ( 2, 3, 4, 10, 100, 1 );
```

```
6   my $smallest= &MIN ( 200, 2, 12, 40, 2, 20 );
7   print "The biggest is $biggest and the smallest is $smallest.\n";

(Output)
    3
    4
    10
    100
    100
    ------------------------------
    200
    2
    2
    2
    2
    ------------------------------
7   The biggest is 100 and the smallest is 2.
```

解释

1. 标量 $max 保存着数组 @_ 的第一个元素值。my 函数使 $max 成为该子例程的局部变量。即使在子例程中修改变量 $max，也不会影响原变量值。
2. 对列表中的每个元素，通过循环将其元素值赋给标量 $foo。
3. 如果 $max 小于 $foo，则让 $max 获得 $foo 变量值。
4. 由于子例程 MAX 的最后一个语句是 $max，因此返回 $max 的值，并在第 5 行赋值给 $biggest。
5. 把 MAX 子例程中最后一个表达式的值赋给标量 $biggest。
6. 把 MIN 函数的返回值赋值给标量 $smallest。这里在子例程 MIN 中显式地用到了 return 函数。

11.2.3 上下文和子例程

在讨论变量和运算符时，本书就引入了“上下文（Context）”的概念。这里将介绍如何在子例程中使用上下文。Perl 主要提供了两种上下文：标量型上下文和列表型上下文。在不同的上下文类型中，表达式的取值结果也将表现为相应的类型。

数组赋值和标量赋值操作提供了使用上下文的典型场景。读者不妨先看下面这个示例：

```
@list = qw( apples pears peaches plums );  # List context
$number = @list;   # Scalar context
```

在列表上下文中，程序将把 @list 赋值为一组数组元素；而在标量上下文中，程序则会将 $number 赋值为数组 @list 的长度值。

此外，用户还会在使用 Perl 内建函数时接触到上下文。不妨以 localtime 函数为例。如果程序将其返回值赋予一个标量，则函数会把日期和时间值以字符串的形式返回；但是如果将其返回值赋予一个数组的话，则函数会返回一个数组，其每个数组元素分别是代表时、分、秒的数字。另一个示例是 print 函数，该函数期望在列表上下文中获得一系列参数。用户可以借助内建的 scalar 函数显式地在标量上下文中计算表达式的值，正如示例 11.5 所示。

示例 11.15

```
    # Context
@now = localtime;  # List context
```

```
2 print "@now\n";    # Scalar context

3 $now = localtime;
4 print "$now\n";

5 print localtime, "\n"; # prints in list context
6 print scalar localtime,"\n"; # Forced to scalar context
(Output)
2   30 40 9 23 3 107 1 112 1
4   Mon Apr 23 09:40:30 2007
5   2021123310711121
6   Mon Apr 23 11:02:20 2007
```

示例 11.16

```
    # Context
1 print "What is your full name? ";

2 ($first, $middle, $last)=split(" ", <STDIN>);# STDIN scalar context

3 print "Hi $first $last.\n";

(Output)
2 What is your full name? Daniel Leo Stachelin
3 Hi Daniel Stachelin.
```

wantarray 函数和用户自定义子例程。 当大家正在为某个故事、圣经问题或政治演说争论不休时，可能会说：“他说的话根本文不对题（out of context）……”。在第 5 章中，我们已经讨论了有关上下文的知识。在 Perl 中，上下文决定了以何种方式对变量或表达式求值。例如：上下文究竟是列表性的还是标量性的？在很多时候，用户往往期望根据调用子例程的上下文类型来决定子例程的行为模式。这时便可使用内建的 wantarray 函数。通过该函数，用户便可判断子例程的返回值类型是列表还是标量。如果在列表上下文中调用子例程（譬如将返回值赋值给一个数组）的话，wantarray 函数会返回 true ；反之则返回 false。如果上下文返回值为空（即空值上下文），则 wantarray 函数将返回未定义值。

示例 11.17

```
#!/usr/bin/perl

  print "What is your full name? ";
  chomp($fullname=<STDIN>);

1 @arrayname = title($fullname); # Context is array
  print "Welcome $arrayname[0] $arrayname[2]!\n";
  print "What is the name of that book you are reading? ";
  chomp($bookname=<STDIN>);
2 $scalarname = title($bookname);  # Context is string
  print "The book $arrayname[0] is reading is $scalarname.\n";

3 sub title{
     # Function to capitalize the first character of each word
     # in a name and to return a string or an array of words
4    my $text=shift;
```

```
     my $newstring;
5    my$text=lc($text);
6    my @newtext=split(" ", $text); # Create a list of words
     foreach my $word ( @newtext ){
        $word = ucfirst($word);  # Capitalize the first letter
7       $newstring .= "$word "; # Create a title string
     }
     @newarray = split(" ", $newstring);
8    # Split the string into an array
     chop($newstring);  # Remove trailing whitespace

9    return wantarray ? @newarray : $newstring;  # Return either array
     # or scalar based on how the subroutine was called

  }
(Output)
  What is your full name? robert james taylor
  Welcome Robert Taylor!
  What is the name of that book you are reading? harry potter half
blood prince
  The book Robert is reading is Harry Potter Half Blood Prince.
```

11.3 按引用调用

11.3.1 符号引用－typeglob

定义。typeglob 是变量的别名，即变量的另一个名字。它又称作是“符号引用”，类似于 UNIX 文件系统中的软链接。如要创建别名，可在实际变量的前面加上一个“*”符号。星号（“*”）适用于任意类型的变量，包括标量、数组、散列、文件句柄和子例程等。别名是符号表中针对同名标识符的另外一个名称。“typeglob”这个名字的来历是：它能以同样的名字表达不同的数据类型。例如，*name 既可以表示 $name、@name，也可表示 %name、&name 等等。

别名机制常常出现在早期（Perl 4）的脚本中，负责按引用进行参数传递。虽然现在 Perl 提供了改进的硬引用机制（详见第 13 章“这个功能需要用到引用吗？”），但用户仍可按照以往的方式使用 typeglob 或别名机制。鉴于早期的 Perl 引入了大量含有 typeglob 机制的库，而且目前在为程序构建符号表时也还需要用到它们（详见第 12 章），因此本章还将对它们进行详细介绍。（如需了解如何在子例程中使用硬引用机制，请阅读“硬引用－指针”一节，完整内容请参阅第 13 章）。

通过别名按引用传递。 用户可将别名（或 typeglob）传递到函数中，以便真正支持按引用传递。这样便可修改全局变量的值，而不是修改 @_ 数组中的局部副本。如需将一个或多个数组传入某个函数的话，用户无需把数组的全部内容一一传入子例程，而是可以传递该数组的别名或指针（详见“硬引用－指针”一节）。如要为某个变量创建相应的别名，请在别名的名称前面加上一个星号（*）字符，如下所示：

```
*alias = *variable ;
```

星号前缀代表所有的变量类型，包括子例程、文件句柄和格式。Typeglob 将自动产生具有相同

名称的标量值，也就是说，它会“glob”到符号表中具有相同名字的所有符号上去[⊖]。这里由程序员自行决定别名应该引用哪种符号。为此在访问别名值时，读者应当在别名名称前面加上正确的前缀字符。例如：

```
Given: *alias = *var
Then: $alias refers to the scalar $var
  @alias refers to the array @var
  $alias{string} refers to an element of a hash %var
```

如果把一个文件句柄传递给子例程，也可通过 typeglob 实现文件句柄的本地化。

Perl 5 改进了原有的别名机制。现在，别名可以仅代表单个特殊字符，而不是所有的特殊字符。此外它还引入了另一种更为方便的按引用传递的方式，称为硬引用（hard reference），该机制类似于 C 语言中的指针。

使别名私有化——local 与 my。 通过 my 函数创建的变量名称并不保存到符号表中，而是位于临时缓冲区里。my 函数能够创建专用于其代码块的私有变量。由于 typeglob 仅能关联到特定包的符号表上，因此 my 函数不能对它进行私有化。如要让 typeglob 本地化，必须使用 local 函数。

示例 11.18

```
(The Script)
    #!/usr/bin/perl
1   $colors="rainbow";
2   @colors=("red", "green", "yellow" );
3   &printit(*colors);            # Which color is this?
4   sub printit{
5       local(*whichone)=@_;      # Must use local, not my with globs
6       print *whichone, "\n";       # The package is main
7       $whichone="Prism of Light";  # Alias for the scalar
8       $whichone[0]="PURPLE";       # Alias for the array
    }
9   print "Out of subroutine.\n";
10  print "\$colors is $colors.\n";
11  print "\@colors is @colors.\n";

Output)
6   *main::colors
9   Out of subroutine.
10  $colors is Prism of Light.
11  @colors is PURPLE green yellow.
```

解释

1. 将标量 $colors 赋值为 rainbow。
2. 向数组 @colors 赋予三个值：red、green 和 yellow。
3. 调用子例程 printit。将所有别名为 colors 的符号作为参数传递给该子例程。这里通过星号创建别名（typeglob）。
4. 定义子例程 printit。
5. 数组 @_ 中含有需要传递的别名。这里使用本地别名 *whichone 向它赋值。*whichone 现在是 colors 符号的另一个别名。

[⊖] 这和文件名替换时执行的“glob”操作（如 <p*>）不同。

6. 如果在这里打印别名本身的值，则只能说明该别名位于 main 包中，以及所有名为 colors 的变量、子例程和文件句柄的符号。
7. 向该别名代表的标量赋予新的值。
8. 针对该别名代表的数组，为其第一个元素赋予新的值。
9. 退出子例程。
10. 退出子例程。此时，标量 $colors 的值已经改变。
11. 退出子例程。此时，数组 @colors 的值已经改变。

示例 11.19

```
(The Script)
    # Revisiting Example 11.6 -- Now using typeglob
1   print "Give me 5 numbers: ";
2   @n = split(' ', <STDIN>);
3   &params(*n);
4   sub params{
5       local(*arr)=@_;
6       print 'The values of the @arr array are ', @arr, "\n";
7       print "The first value is $arr[0]\n";
8       print "the last value is ", pop(@arr), "\n";
9       foreach $value(@arr){
10          $value+=5;
11          print "The value is $value.\n";
        }
    }
    print "Back in main\n";
12  print "The new values are @n.\n";

(Output)
1   Give me 5 numbers: 1 2 3 4 5
6   The values in the @arr array are 12345
7   The first value is 1
8   The last value is 5
11  The value is 6
    The value is 7
    The value is 8
    The value is 9
    Back in main
12  The new values are 6 7 8 9  <--- Look here. Got popped this time!
```

解释

1. 要求用户提供输入。
2. 基于空白字符对用户输入内容进行拆分（split），并返回给数组 @n。
3. 调用子例程 params。将符号表中对应于 n 的任意一个别名作为参数传递给该子例程。
4. 定义子例程 params。
5. 在子例程 params 中，将别名传递给 @_ 数组。将该值赋予本地别名（typeglob）*arr。
6. 打印数组 @arr 的值。请记住：@arr 只是数组 @n 的一个别名，引用 @n 数组中的元素值。
7. 打印数组的第一个元素。
8. 弹出数组中的最后一个元素，而不只是引用它。
9. foreach 循环依次把数组 @arr 中的每个元素赋值给标量 $value。

10. 将数组中的每个元素递增 5，并保存到标量 $value。
11. 打印新的值。
12. 打印出来的值表明：该函数通过别名修改的值真的在数组 @n 中改变了。请参考示例 11.6。

通过引用传递文件句柄。　如要直接把文件句柄传递给子例程，惟一的途径就是通过引用。用户可以使用 typeglob 为文件句柄创建别名或使用硬引用（有关硬引用的更多内容，详见第 13 章）。

示例 11.20

```
(The Script)
    #!/bin/perl
1   open(READMEFILE, "f1") || die;
2   &readit(*READMEFILE);      # Passing a filehandle to a subroutine
    sub readit{
3       local(*myfile)=@_;     # myfile is an alias for READMEFILE
4       while(<myfile>){
           print;
        }
    }
```

解释

1. open 函数以读方式打开 UNIX 文件 f1，并将其附加给文件句柄 READFILE。
2. 调用子例程 readit。这里通过 typeglob 为文件句柄指定一个别名，并将其作为参数传递给子例程。
3. 将 @_ 的值赋予本地别名 myfile，即将别名传递给子例程。
4. 别名是文件句柄 READFILE 的另一个名字，位于角括号之间。然后通过 while 循环逐行读取文件句柄中的各行内容，并打印之。

选择性别名与反斜杠运算符。 Perl 5 的引用机制允许对某类特定变量而不是所有变量类型使用别名。例如：

```
*array=\@array;
*scalar=\$scalar;
*hash=\%assoc_array;
*func=\&subroutine;
```

示例 11.21

```
(The Script)
    # References and typeglob
1   @list=(1, 2, 3, 4, 5);
2   $list="grocery";
3   *arr = \@list;       # *arr is a reference only to the array @list
4   print @arr, "\n";
5   print "$arr\n";          # Not a scalar reference
    sub alias {
6       local (*a) = @_;     # Must use local, not my
7       $a[0] = 7;
8       pop @a;
    }
9   &alias(*arr);        # Call the subroutine
```

```
10  print "@list\n";
11  $num=5;
12  *scalar=\$num;      # *scalar is a reference to the scalar $num
13  print "$scalar\n";

(Output)
4   1 2 3 4 5
5
10  7 2 3 4
13  5
```

解释

1. 将一个列表赋值给数组 @list。
2. 为标量 $list 赋值。
3. 别名 *arr 是数组 @list 的另一个名字。它不会是任何其他类型的别名。
4. 通过别名 *arr 引用数组 @list。
5. 别名 *arr 不引用变量。这里不会打印任何内容。
6. 在子例程中，局部变量 *a 接收来自参数传递的别名的值，并赋值给 @_ 数组。
7. 通过别名为数组赋予新的值。
8. 通过别名弹出数组的元素值。
9. 调用子例程，并把别名 *arr 作为参数传递。
10. 打印 @list 的值，表明子例程所作的修改内容。
11. 给标量 $num 赋值。
12. 创建一个新别名。*scalar 只引用标量 $num。
13. 别名只是标量 $num 的另一个名字。打印其值。

11.3.2 硬引用－指针

在按引用向子例程传递参数时，目前比 typeglobs 机制更常用到的途径是通过指针（pointer）。在作更进一步的介绍之前，本节将首先定义指针及其格式，还有如何使用指针。然后提供一些实例。有关指针使用方面的更多知识，请阅读第 13 章。

定义。 硬引用（hard reference），通常又称为指针（pointer），是含有其他变量地址的标量型变量。用户可通过反斜杠（\）运算符来创建指针。在打印指针值时，用户不但可以看到保存在指针中的十六进制地址，还能观察到驻留于该地址的数据类型。

例如，如果编写如下代码：

```
$p = \ $name ;
```

则会把标量 $name 的地址赋值给 $p。$p 就是对标量 $name 的引用。在打印时，$p 变量值的格式为 SCALAR（0xb057c）。

由于指针中含有地址信息，因此可用于按引用为字符串传递参数。此外，由于指针本质上只是一个标量型变量，而非 typeglob 别名，因此可以通过 my 函数将其声明为专用的词法变量。

```
my $arrayptr=\@array;        # creates a pointer to an array
my $scalarptr=\$scalar;      # creates a pointer to a scalar
my $hashptr=\%assoc_array;   # creates a pointer to a hash
my $funcptr=\&subroutine;    # creates a pointer to a subroutine
```

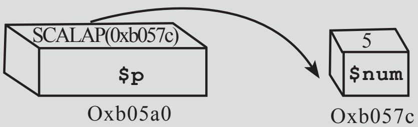
访问指针地址。 如果打印指针的值，就会看到一串地址。如果需要获取该地址上保存的值，即按地址访问指针，则必须使用两个特殊符号作为指针的起始字符。一个是美元符号（$），由于指针本身是标量性的，因此位于指针前面的特殊符号表明了指针的数据类型。例如，如果 $p 是对标量 $x 的引用，则 $$p 就能获得 $x 的值；如果 $p 是对数组 @x 的引用，则 @$p 将得到 @a 的值。在上述两个示例中，位于 $p 前面的特殊符号表明了它所指向变量的数据类型。在更为复杂的例子中，用户还可使用箭头（中缀）运算符（有关箭头运算符的知识请参阅“引用和匿名变量”一节）。表 11-1 给出了一些创建指针和访问指针地址内容的例子。

表 11-1　创建指针和访问指针地址

赋值	创建引用	访问地址	使用箭头运算符访问地址
$sca = 5 ;	$p = \$sca	print $$p;	
@arr = (4,5,6);	$p = \@arr	print @$p;	$p->[0]
		print $$p[0];	
%hash=(key=>'value');	$p = \%hash	print %$p;	$p->{key}
		print $$p{key};	

示例 11.22

```
(The Script)
    #!/bin/perl
1   $num=5;
2   $p = \$num;       # The backslash operator means "adddress of"
3   print 'The address assigned $p is ', $p, "\n";
4   print "The value stored at that address is $$p \n";
```




```
Memory Addresses

(Output)
  3   The address assigned $p is SCALAR(0xb057c)
  4   The value stored at that address is 5
```

解释

1. 把标量 $num 赋值为 5。
2. 把 $num 的地址赋值给标量 $p。反斜杠运算符在这里负责创建引用。$p 又称为引用或指针（两个名称可以互换使用）。
3. 打印保存在 $p 中的地址。Perl 还给出其变量类型是 SCALAR。
4. 如要根据地址访问 $p 的内容，则应给 $p 加上另一个美元符号（$）。这个额外的美元符号告诉 Perl 去查找 $p 引用的变量值，即 $num。

示例 11.23

```
(The Script)
```

```
1   @toys = qw( Buzzlightyear Woody Thomas Pokemon );
2   $num = @toys;
3   %movies=("Toy Story"=>"US",
             "Thomas"=>"England",
             "Pokemon"=>"Japan",
            );
4   $ref1 = \$num;         # Scalar pointer
5   $ref2 = \@toys;        # Array pointer
6   $ref3= \%movies;       # Hash pointer
7   print "There are $$ref1 toys.\n";    # Dereference pointers
8   print "They are: @$ref2.\n";
9   while( ($key, $value) = each ( %$ref3 )){
10     print "$key--$value\n";
    }
11  print "His favorite toys are $ref2->[0] and $ref2->[3].\n";
12  print "The Pokemon movie was made in $ref3->{'Pokemon'}.\n";
(Output)
7   There are 4 toys.
8   They are: Buzzlightyear Woody Thomas Pokemon.
10  Thomas--England
    Pokemon--Japan
    Toy Story--US
11  His favorite toys are Buzzlightyear and Pokemon.
12  The Pokemon movie was made in Japan.
```

解释

1. 将一个列表赋值给数组 @toys。
2. 将标量型变量 $num 赋值给数组 @toys，并返回数组的元素个数。
3. 为散列型变量 %movies 赋予键 / 值对。
4. 引用 $ref1 是一个标量。这里将标量 $num 的地址赋值给它。反斜杠运算符用于创建引用。
5. 引用 $ref2 是一个标量。这里将数组 @toys 的地址赋值给它
6. 引用 $ref3 是一个标量。这里将散列变量 %movies 的地址赋值给它
7. 按地址访问引用，即：跳转到 $ref1 指向的地址，然后打印保存在该地址的标量内容。
8. 按地址访问引用，即：跳转到 $ref2 指向的地址，然后获取数组并打印其内容。
9. 内建的 each 函数负责从散列中获取键和值。散列型指针 $ref3 的前面带有百分号 %，也就是说，本行将根据地址访问散列指针的内容。
10. 从散列变量 %movies 中打印所有键 / 值对。
11. 按地址访问指针，并获得数组的第一个和第四个元素。若要按地址访问数组中的指针，可以使用箭头运算符和数组元素下标。此外还可使用形如 $$ref2[0] 或 $$ref2[3] 的格式，不过这种格式读写起来都不大方便。
12. 通过箭头运算符访问指针地址内容。这里用花括号围住散列值。此外还可使用形如 $$ref3{Pokemon} 的格式。

指针参数。　当一个子例程接收参数时，便会把参数保存到 @_ 特殊数组中。例如，当把两个数组同时传递给一个子例程时，这两个数组都将保存到 @_ 特殊数组中，并连接成单个列表。如果不知道其中前一个数组长度的话，基本上就不可能再拆开这两个数组了。不过，如果传递给子例程的是两个指针，分别含有这两个数组地址的话，就可以方便地实现数组内容的分离，并能按地址访问数组内容了。详见示例 11.24。

示例 11.24

```
 (The Script)
# Passing by reference with pointers
1 @list1= (1..100);
2 @list2 = (5..200);

3 display(@list1, @list2); # Pass two arrays

  print "-" x 35,"\n";

4 display(\@list1, \@list2); # Pass two pointers


5 sub display{
    print "@_\n";
 }

(Output)
 3   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67

                     <continues>

    177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192
193 194 195 196 197 198 199 200
    -----------------------------------
 4   ARRAY(0x182e048) ARRAY(0x182ea38)
```

把指针传递给子例程

示例 11.25

```
(The Script)
   # This script demonstrates the use of hard references
   # when passing arrays. Instead of passing the entire
   # array,a hard reference (pointer) is passed.
   # The value of the last expression is returned.

1  my @list1=(1 .. 100);
2  my @list2=(5,10,15,20);

3  print "The total is : ", &addemup( \@list1, \@list2) ,".\n";
          # two pointers

4  sub addemup{
5      my( $arr1, $arr2) = (shift, shift);       # The two pointers
                                                 # are shifted from @_
6      my $arrl = 0;
7      print $arr ,"\n";
8      print $arr2, "\n";
9      foreach $num ( @$arr1,@$arr2 ){   # dereference the pointers
10        $total+=$num;
       }
11     $total;  # The expression is evaluated and returned
   }

(output)
7  ARRAY(0x8a62d68)
7  ARRAY(0x8a60f2c)
3  The total is:  5100.
```

解释

1. 为数组 @list1 赋予 1 到 100 之间的值。
2. 将四个值赋予数组 @list2。
3. 以两个参数调用子例程 &addemp。本行的反斜杠用于创建指针，以便传递 @list1 和 @list2 的地址。
4. 声明子例程 &addemp。
5. 数组 @_ 中含有刚刚传入的两个参数。将参数从数组 @_ 转移到 my 变量 $arr1 与 $arr2 中，它们都是指针。
6. 为 $total 赋予初始值 0。
7. 打印指针值。该指针指向数组 @list1。
8. 打印指针值。该指针指向数组 @list2。
9. 进入 foreach 循环，通过按地址访问指针的方式依次把 @list1 中的每个元素赋值给 $num，直到数组中所有元素都处理完毕。
10. 将 $num 的每个变量值相加，并赋值给 $total 中，直到循环执行完毕。
11. 在第 3 行返回 $total 的总和。然后将该值作为参数传给 print 函数打印出来。

11.3.3 自动加载

Perl 的 AUTOLOAD 函数能检查是否定义了子例程。当程序要求调用子例程，同时又没有找到该子例程的定义时，就会去调用 AUTOLOAD 子例程。然后还会把未定义的子例程名称赋值给特殊变量 $AUTOLOAD。

此外，AUTOLOAD 函数还可用在对象中，提供了调用匿名方法（anonymous method）的实现机制（方法是面向对象语境中子例程的另一种称呼）。

示例 11.26

```
(The Script)
    #!/bin/perl
1   sub AUTOLOAD {
2      my(@arguments)=@_;
3      $args=join(', ', @arguments);
4      print "$AUTOLOAD was never defined.\n";
5      print "The arguments passed were $args.\n";
    }

6   $driver="Jody";
    $miles=50;
    $gallons=5;

7   &mileage($driver, $miles, $gallons);  # Call to an undefined
                                          # subroutine
(Output)
4   main::mileage was never defined.
5   The arguments passed were Jody, 50, 5.
```

解释

1. 定义子例程 AUTOLOAD。

2. 调用子例程 AUTOLOAD，其参数与第 7 行中调用原有子例程的参数相同。
3. 通过逗号连接各个参数，并将它们保存到标量 $args 中。
4. 把原有的包和调用的子例程名保存到标量 $AUTOLOAD 中（对于本例而言，默认包名是 main）。
5. 打印参数。
6. 对标量型变量进行赋值。
7. 通过三个参数调用子例程 meleage。如果需要调用尚未定义的子例程，Perl 便会自动调用 AUTOLOAD 函数，其传递的参数和传给子例程 mileage 的参数相同。

示例 11.27

```
    #!/bin/perl
    # Program to call a subroutine without defining it
1   sub AUTOLOAD {
2       my(@arguments) = @_;
3       my($package, $command)=split("::",$AUTOLOAD, 2);
4       return '$command @arguments';   # Command substitution
    }

5   $day=date("+%D");      # date is an undefined subroutine
6   print "Today is $day.\n";
7   print cal(3,2007);     # cal is an undefined subroutine
```

```
(Output)
Today is 03/26/07.

     March 2007
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

解释

1. 定义子例程 AUTOLOAD。
2. 调用子例程 AUTOLOAD，其参数与第 5 行、第 7 行中调用原有子例程的参数相同。
3. 借助双冒号（::）定界符将变量 AUTOLOAD 的内容拆分（split）为两个部分。返回的数组含有两个元素：包名（package name）和调用的子例程名。
4. 返回值是调用函数的名称，在第一种情况下是一个 UNIX 命令及其参数，这里的反引号将使其中的字符串以 UNIX 命令格式执行。
5. 本示例没有定义子例程 date。因此 AUTOLOAD 将找到其名字，并在 AUTOLOAD 函数中将它赋值给 $AUTOLOAD。函数 date 接收一个参数，即参数 +%D。该参数也是 UNIX 命令 date 支持的参数，它表示返回当前的日期值。
6. 打印返回值内容。
7. 本示例没有定义子例程 cal。它接收两个参数。AUTOLOAD 将 cal 赋值给 $AUTOLOAD。其参数是赋予数组 @arguments 的 3 和 2003。将参数传递给 AUTOLOAD 函数，并在第 4 行使用它。在变量替换完毕后，反引号将执行位于其中的字符串。即：执行 UNIX 命令 cal 3 2003，并将结果返回给 print 函数。

11.3.4 BEGIN 和 END 子例程（开始与结束）

这里的 BEGIN 和 END 函数会让 UNIX 程序员回忆起 awk 编程语言中的 BEGIN 和 END 模式。对于 C++ 程序员而言，可以把 BEGIN 看作是构造函数，并把 END 看作是析构函数。BEGIN 和 END 函数的功能与它们相似。

BEGIN 子例程将在程序开头立刻执行，甚至优先于程序中其他部分的解析工作。如果提供了多个 BEGIN 函数的话，则按照它们定义的顺序依次执行。

当所有任务完成之后，程序便会执行 END 子例程。甚至在通过 die 函数退出程序时也是如此。多个 END 函数将按相反的顺序予以执行。

在使用这些特殊子例程时，用户必须提供关键字 sub。

示例 11.28

```
    #!/bin/perl
    # Program to demonstrate BEGIN and END subroutines
1   chdir("/stuff") || die "Can't cd: $!\n";
2   BEGIN{ print "Welcome to my Program.\n"};
3   END{ print "Bailing out somewhere near line ",__LINE__,
                                              " So long.\n"};

(Output)
Welcome to my Program.
Can't cd: No such file or directory
Bailing out somewhere near line 5. So long.
```

解释

1. 尝试把目录改名为 /stuff。这里 chdir 执行失败，因此会调用 die 函数。在一般情况下，程序会立刻退出。但是这里定义了一个 END 子例程，因此在退出程序前必须首先执行 END。
2. 尽快执行 BEGIN 子例程，即在定义后马上执行它。该子例程将在程序其他部分执行之前就予以执行。
3. 在退出程序时，始终需要执行 END 子例程，即便在调用 die 函数时也是如此。这里只是为 awk 程序员打印一行信息。

11.3.5 subs 函数

subs 函数用于预先声明子例程。其参数是子例程的列表。这样一来，即便不用 & 和括号，用户也能调用子例程，并能覆盖内建的 Perl 函数。

示例 11.29

```
    #!/bin/perl
    # The subs module
1   use subs qw(fun1 fun2 );

2   fun1;
3   fun2;

4   sub fun1{
       print "In fun1\n";
```

```
    }

5   sub fun2{
       print "In fun2\n";
    }

(Output)
In fun1
In fun2
```

解释

1. 加载 sub 模块（详见“use 函数（模块与程序）”一节），并给出子例程列表。
2. 既不通过 &，也不使用括号，调用子例程 fun1，因为它位于 subs 列表中。之后定义该函数。
3. 在定义之前调用 fun2。

11.4 读者应当学到的知识

1. 如何定义和调用子例程（subroutine）？
2. 函数（function）和子例程之间有什么差别？
3. 在 Perl 脚本中，可以将子例程的定义放在哪里？
4. 如何把参数传递给子例程？
5. Perl 是如何检索参数列表的？
6. 全局变量和局部变量有什么异同？
7. 按值传递和按引用传递方式有什么不同？
8. 硬引用（hard reference）的另一个名称是什么？
9. 什么是 typeglob ？
10. return 语句有什么重要性？
11. 什么是原型（prototype）？
12. 什么是自动加载（autoloading）？

11.5 下章简介

在下一章内容中，读者将扬帆远航，从“入门级”Perl 程序员迈向“高级”程序员水平。我们将不再编写孤立的小脚本，而是开始学习如何使用 Perl 业已提供的库和模块。读者将浏览 CPAN 内容，并学习如何下载和使用其他程序员编写好的模块。

此外，读者还将了解有关包（package）和命名空间（namespace）、如何导出（export）/ 导入（import）符号、如何使用 Perl 标准库、以及如何创建自定义库等方面的知识。最后还会学习如何创建过程模块（procedural module），以及如何保存并使用这些模块。

练习 11 不可或缺的子例程

1. 编写一段名叫 tripper 的程序，要求用户输入驾驶里程和耗用的汽油数目。在 tripper 中，编写一个子例程 mileage，计算用户驾驶的耗油率（每加仑油的公里数）。该程序的参数是行驶里程数和

耗用的汽油量。所有变量都应当是 my 变量。最后打印计算结果。在程序中应当提供 tripper 的原型。

2. 酒店通常都是根据代表其等级的星级情况来收费的。一个五星级酒店里可能有特大床、厨房、两个电视；而一星级酒店中就可能只有蟑螂和漏水的屋顶。编写一个名叫 printstar 的子例程，生成下列散列变量中各酒店的星级收费结果。printstar 函数接受两个参数：酒店名字和星级数。（小提示：可以将散列的键排列为数组，然后循环遍历所有的键，并对每一次循环调用 printstar 函数）。

```
%hotels=("Pillowmint Lodge" => "5",
         "Buxton Suites"    => "5",
         "The Middletonian" => "3",
         "Notchbelow"       => "4",
         "Rancho El Cheapo" => "1",
         "Pile Inn"         => "2",
        );

(OUTPUT)
Hotel                    Category
------------------------------------------
Notchbelow           |****          |
The Middletonian     |***           |
Pillowmint Lodge     |*****         |
Pile Inn             |**            |
Rancho El Cheapo     |*             |
Buxton Suites        |*****         |
------------------------------------------
```

将酒店按星级排序，五星级最前，一星级最后。

3. 编写一段名为 grades 的程序，以课程编号和学生姓名作为参数。课程编号包括 CS101、CS202 和 CS303。该程序包含三个子例程：

   a. 子例程 ave 负责计算一组成绩的平均值。

   b. 子例程 highest 负责计算该组成绩的最大值。

   c. 子例程 lowest 负责计算该组成绩的最小值。

   打印平均值、最大值和最小值。如有不及格（平均值低于 60）的情况，则向 STDERR 打印该同学的姓名、课程编号和相应的警告信息，如：Be advised：Joe Blow failed CS202。然后，把不及格同学的姓名与课程编号发送到文件 failures 中，并按照成绩顺序进行排序。

   使用 AUTOLOAD 函数检查每个子例程是否已经定义。

第 12 章　模块化、打包并发送到库

12.1　包和模块

12.1.1　引言

在下面几节中，我们将讨论标准 Perl 库中的包和模块以及如何使用它们。目前许多模块都使用面向对象的方法进行程序设计，本书将在第 14 章中予以讨论，并介绍诸如类、对象和方法之类的术语。本章重点介绍面向函数的模块和库，以及如何使用与创建它们。本章还将介绍如何从 CPAN（Comprehensive Perl Archive Network，Perl 综合文档网）获取模块，在第 14、15 和 18 章中将更详细地介绍如何安装和使用 CPAN 模块。

12.1.2　一个类比

两个孩子都有一盒乐高积木玩具。其中一套乐高积木可以建一艘船，另一套可以建一架飞机。这两个孩子打开盒子并把其中的积木撒在地板上，它们混在一起。乐高积木块具有不同的形状和颜色。两个盒子中都有黄色的方块、红色的三角块和蓝色的矩形块，但是它们现在混在了一起，很难分清哪些应该用于造飞机，哪些应用于造船。如果这些积木都放在它们单独的盒子里，那就始终不会出现这种混淆情况了。

在 Perl 中，这些单独的盒子称为包（package），乐高积木块则称为符号（symbol）；即变量和常量的名称。通过把符号保存在它们自己的私有包中，用户便可在程序中导入库模块和例程，从而避免变量名与所包含的模块或库文件中的命名发生冲突。

12.1.3　定义

将数据和函数包装到单独的命名空间中称为封装（encapsulation）（C++ 程序员称之为类（class），面向对象的 Perl 程序员也称之为类）。单独的命名空间又称作是包（package）。单独的命名空间意味着：对于命名包中所有的变量，Perl 都持有一个单独的符号表。默认的当前包是 main 包。迄今为止所有的示例脚本都位于包 main 中。在默认情况下，包内所有的变量都是全局变量。包机制允许用户切换命名空间，从而让包中的变量成为私有变量，即使它们在包外拥有相同的名称也

可如此（参见图 12-1）。

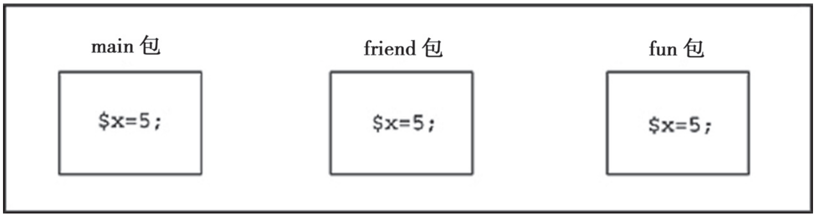


图 12-1　每个包都有它自己的命名空间（符号表）

包的作用域是从声明位置一直到文件末尾、最内层封闭块的末尾，或者直到另一个包的声明为止。包的作用域通常就等于文件的作用域。如要引用另一个包中的变量，用户可以在包名前面加上代表变量数据类型的特殊字符作为前缀，随后提供双冒号和变量名。在老版本的 Perl 4 中，可使用撇号代替上述冒号[⊖]（双冒号令人想起 C++ 中的作用域解析运算符）。在引用 main 包时，还可以删除包名。

Perl 5 将包的概念扩展成了模块（module）。模块通常是在库中定义且可重用的包。模块要比简单的包更为复杂。它们能够把符号导出到其他包中，并且可以与类和方法协同工作。模块是一种保存在文件中的包，只需在文件名后面追加 .pm 扩展名就构成了模块名。Use 函数以模块名作为参数，负责把该模块加载到脚本中。

```
$package 'variable
$package::variable
$main::variable
$::variable
```

12.1.4　符号表

在编译程序时，编译器必须记录使用的所有变量名、文件句柄、目录句柄、格式和子例程。Perl 将这些符号的名称作为键存储在每个包的散列表中。散列的名称与包名相同。与散列键相关联的值是对应的 typeglob 值，又称为别名（alias）。typeglob 会“glob”所有可用符号名称表达的类型（参见第 11 章中关于别名和 typeglob 的介绍）。对于用相同名称表示的每个值，Perl 实际上会创建单独的内部指针（参见图 12-2）。

每个包都有它自己的符号表。无论何时使用包声明，都要切换到该包的符号表。

用户可通过双冒号（::）用包名限定某个由 local 函数赋值的变量，从而实现在另一个包中访问该变量。此时该变量仍在作用域内，并可从主符号表中访问它。

由 my 函数赋值的变量将不能从其所在包的外面访问。它们并没有存储在包符号表中，而是存储在为每个调用子例程创建的专用缓冲区里。因此，在使用“my”变量时，用户将无法通过包的符号表来访问它们，因为它们根本不在那里！

在下面的示例中，读者将注意到 main 包不但保存了程序提供的符号，还保存了其他一些符号，如 STDIN、STDOUT、STDERR、ARGV、ARGVOUT、ENV 和 SIG 等。这些符号和特殊变量（如 $_ 和 $!）都是强制保存在包 main 里的。除非进行限定，否则其他包也可以引用这些符号。示例 12.1 显示了 main 包的符号表的内容。

⊖　从版本 5.003 起，在 Perl 5 脚本中仍然接受撇号。

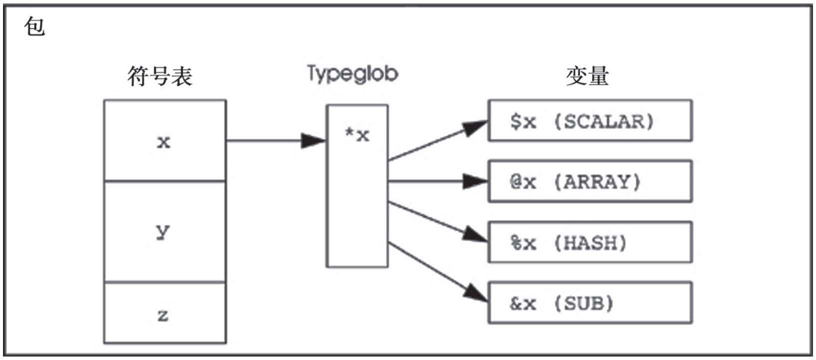


图 12-2 每个符号都被赋予一个 typeglob（即 *x），它表示所有名为 x 的类型

示例 12.1

```
(The Script)
    #!/bin/perl
    # Package main
    use strict "vars";
    use warnings;
    our ( @friends, @dogs, $key, $value );    # Declaring variables
    my($name,$pal,$money);
    $name="Susanne";
    @friends=qw(Joe Jeff Jan );
    @dogs = qw(Guyson Lara Junior);
    local $main::dude="Ernie";    # Keep strict happy
    my $pal = "Linda";            # Not in the symbol table
    my $money = 1000;
    while(($key, $value) = each (%main::)){
                                 # Look at main's symbol table
        print "$key:\t$value\n";
    }

(Output)
Name "main::dude" used only once: possible typo at packages line 10.
STDOUT:             *main::STDOUT
@:                  *main::@
ARGV:               *main::ARGV
STDIN:              *main::STDIN
:                   *main::
dude:               *main::dude
attributes:::       *main::attributes::
DB:::               *main::DB::
key:                *main::key
_<..\xsutils.c:     *main::_<..\xsutils.c
_<perllib.c:        *main::_<perllib.c
UNIVERSAL:::        *main::UNIVERSAL::
?:                  *main::?
value:              *main::value
DynaLoader:::       *main::DynaLoader::
?:                  *main::?
?ARNING_BITS:       *main::?ARNING_BITS
SIG:                *main::SIG
Exporter:::         *main::Exporter::
```

```
Win32:::             *main::Win32::
warnings:::          *main::warnings::
BEGIN:               *main::BEGIN
stderr:              *main::stderr
INC:                 *main::INC
_:                   *main::_
":                   *main::"
DATA:                *main::DATA
_<.\win32.c:         *main::_<.\win32.c
$:                   *main::$
stdout:              *main::stdout
IO:::                *main::IO::
ENV:                 *main::ENV
dogs:                *main::dogs
strict:::            *main::strict::
stdin:               *main::stdin
Carp:::              *main::Carp::
CORE:::              *main::CORE::
/:                   *main::/
0:                   *main::0
friends:             *main::friends
_<..\universal.c:    *main::_<..\universal.c
STDERR:              *main::STDERR
main:::              *main::main::
```

解释

1. 除非有全局变量，否则会打开 strict 编译指示（pragma）。
2. 如果有裸字（bareword）、未初始化的变量等，warnings 编译指示就会发出警告。
3. 声明一个全局的词法变量 out。strict 编译指示将不会把它们当作全局变量。
4. 声明具有词法作用域的变量 my。
5. 给 my 变量 $name 赋值 Susanne。
6. 赋予数组 @friends 一个名字列表。
7. 赋予数组 @dogs 一个列表。
8. 为了防止 strict 编译指示报错，必须使用包名来完全限定 local 变量 $dude。
9. 给词法作用域内的 my 变量赋值。该变量将不会出现在符号表中。它将保存在代码块的特殊缓冲区里。
10. 这个 my 变量也不会出现在符号表中。
11. 打印 main 符号表的键和值。键是标识符的名称，值是对应的 typeglob 值。

示例 12.2

```
(The Script)
    # Default package is main
    @town = qw(Boston Chico Tampa);
    $friend="Mary";

    print "In main: \$friend is $friend\n";
    package boy;           # Package declaration
    $name="Steve";
    print "In boy \$name is $name.\n";
    $main::friend="Patricia";
    print "In boy \@town is @::town\n";
```

```
9   package main;          # Package declaration
10  print "In main: \$name is $name\n";
11  print "In main: \$name is $boy::name\n";
12  print "In main: \$friend is $friend.\n";

(Output)
3   In main: $friend is Mary
6   In boy $name is Steve.
8   In boy @town is Boston Chico Tampa
10  In main: $name is
11  In main: $name is Steve
12  In main: $friend is Patricia
```

解释

1. 默认包名是 main。在 main 包中为数组 @town 赋值，@town 位于 main 包的符号表中。
2. 在包 main 中把标量 $friend 赋值为 Mary。$friend 位于 main 包的符号表中。
3. 打印 $friend 的值。
4. 声明包 boy。现在从包 main 切换到包 boy。该包具有它自己的符号表。从这一刻开始直到程序结束或声明其他包为止，boy 包都在其作用域中。在这个包中创建的变量是全局变量，但不属于 main 包的一部分。
5. 给标量 $name 赋值 Steve。
6. 打印 $name 中的值。
7. 如要访问包 main 中的变量 $friend，首先应当在包名前面加上美元符号，表明变量类型是标量型，其后则是包名 main、双冒号、变量名 friend。这样就能允许用户在包内切换命名空间。将 Patricia 赋值给 main 包内的变量 $friend。
8. 由于 main 是默认包，切换命名空间时无需使用其名称。@::town 告诉 Perl 将命名空间切换回 main 包中的数组 @town。
9. 声明包 main。现在切换回 main 包中。
10. 在 main 包中，没有定义 $name。
11. 打印 boy 包内 $name 变量的值。
12. boy 包中的 $friend 标量发生变化，这是因为包 main 完全限定了它; 也就是说，当给该变量赋值时，程序将临时切换到包 main。打印这个变量的值。

示例 12.3

```
(The Script)
    # Package declarations
1   $name="Suzanne";      # These variables are in package main
2   $num=100;
3   package friend;       # Package declaration
4   sub welcome {
5       print "Who is your pal? ";
6       chomp($name=<STDIN>);
7       print "Welcome $name!\n";
8       print "\$num is $num.\n";     # Unknown to this package
9       print "Where is $main::name?\n\n";
    }
10  package main;         # Package declaration; back in main
11  &friend::welcome;     # Call subroutine
12  print "Back in main package \$name is $name\n";
```

```
13  print "Switch to friend package, Bye ",$friend::name,"\n";
14  print "Bye $name\n\n";

15  package birthday;      # Package declaration

16  $name="Beatrice";
17  print "Happy Birthday, $name.\n";
18  print "No, $::name and $friend::name, it is not your birthday!\n";

(Output)
5   Who is your pal? Tommy
7   Welcome Tommy!
8   $num is .
9   Where is Suzanne?

12  Back in main package $name is Suzanne
13  Switch to friend package, Bye Tommy
14  Bye Suzanne

17  Happy Birthday, Beatrice!!
18  No, Suzanne and Tommy, it is not your birthday!
```

解释

1. 为默认包 main 给标量 $name 赋值 Suzanne。
2. 给标量 $num 赋值 100。
3. 声明包 friend。在第 10 行以前，它是在作用域内的。
4. 在 friend 包内定义子例程 welcome。
5. 要求用户输入。
6. 给 $name 赋值。包名是 friend。
7. 打印 $name 的值。
8. 标量 $num 是 main 包中的局部变量；在这里没有定义它。没有打印内容。
9. 如要从包 main 访问变量 $name，可在包名 main 后面加上双冒号和变量名。请注意，$ 是位于包名前面的，而不是变量前面。
10. 在子例程外面声明包 main。
11. 除非使用包名予以限定，否则不能调用子例程 welcome。
12. 包 main 中的 $name 值为 Suzanne。
13. 要从包 friend 访问变量，变量名前必须有包名。这将导致切换命名空间。
14. 包 main 中的 $name 值为 Suzanne。
15. 声明新包 birthday。它在块结束前保持有效——此处也就是在脚本结束前。
16. 给包 birthday 中的 $name 变量赋值。
17. 在包 birthday 中，$name 是 Beatrice。
18. 为了从以前的包中访问变量，变量名前必须有包名和双引号。

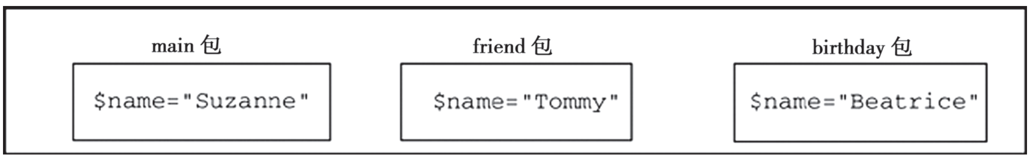


图 12-3　包负责创建单独的命名空间，如示例 12.3 所示

12.2 标准 Perl 库

Perl 发行包中带有很多标准的 Perl 库函数和包。Perl 4 中的库例程是一组过程性程序，其名字都以 .pl 扩展名结尾。Perl 5 中的模块则以 .pm 扩展名结尾。在 Perl 5 中，.pm 文件又称为模块（module）。.pm 文件是以两种不同的编程模式编写的模块：过程的（procedural）和面向对象的（object oriented）。模块的文件名一般以大写字母开头；而那些以小写字母开头的 .pm 文件则是特殊类型模块，又称作编译指示（programa）。编译指示是一种特殊的模块，负责告诉编译器在编译 Perl 程序时必须检查哪些条件。没有扩展名的文件则属于子目录。它们各自含有几个 .pm 文件，划分为一些常用模块。例如，Math 子目录中含有 BigFloat.pm、BigInt.pm、Complex.pm 以及 Trig.pm。

下面是一些标准的 Perl 库内容㊀：

```
AnyDBM_File.pm   Exporter.pm      Sys              complete.pl      open3.pl
AutoLoader.pm    ExtUtils         Term             constant.pm      ops.pm
AutoSplit.pm     Fatal.pm         Test             ctime.pl         overload.pm
B                Fcntl.pm         Test.pm          diagnostics.pm   perl5db.pl
B.pm             File             Text             dotsh.pl         perllocal.pod
Benchmark.pm     FileCache.pm     Thread           dumpvar.pl       ppm.pm
ByteLoader.pm    FileHandle.pm    Thread.pm        exceptions.pl    pwd.pl
CGI              FindBin.pm       Tie              fastcwd.pl       re.pm
CGI.pm           Getopt           Time             fields.pm        shellwords.pl
CORE             I18N             UNIVERSAL.pm     filetest.pm      sigtrap.pm
CPAN             IO               User             find.pl          stat.pl
CPAN.pm          IO.pm            XSLoader.pm      finddepth.pl     strict.pm
Carp             IPC              abbrev.pl        flush.pl         subs.pm
Carp.pm          Math             assert.pl        ftp.pl           syslog.pl
Class            Net              attributes.pm    getcwd.pl        tainted.pl
Config.pm        O.pm             attrs.pm         getopt.pl        termcap.pl
Config.pm~       Opcode.pm        auto             getopts.pl       timelocal.pl
Cwd.pm           POSIX.pm         autouse.pm       hostname.pl      unicode
DB.pm            POSIX.pod        base.pm          importenv.pl     utf8.pm
Data             Pod              bigfloat.pl      integer.pm       utf8_heavy.pl
Devel            SDBM_File.pm     bigint.pl        less.pm          validate.pl
DirHandle.pm     Safe.pm          bigrat.pl        lib.pm           vars.pm
Dumpvalue.pm     Search           blib.pm          locale.pm        warnings
DynaLoader.pm    SelectSaver.pm   bytes.pm         look.pl          warnings.pm
English.pm       SelfLoader.pm    bytes_heavy.pl   network.pl
Env.pm           Shell.pm         cacheout.pl      newgetopt.pl
Errno.pm         Socket.pm        charnames.pm     open.pm
Exporter         Symbol.pm        chat2.pl         open2.pl
```

12.2.1 @INC 数组

特殊数组 @INC 含有指向库例程所在位置的目录路径。如需包含 @INC 数组中没有的目录，则可在命令行中使用 -l 开关㊁，或将环境变量 PERL5LIB 设置为其全路径名。一般应当把这个变量设置到初始化文件中去。如果使用的是 UNIX，初始化文件就是 .origin 或 .profile 文件；如果使用 Windows 的话，则如图 12-4 所示。

㊀ 可在安装 Perl 时确定指向标准 Perl 库的路径名。可以将它设定为默认值，亦可由 Perl 安装人员负责指定。

㊁ 有关 -l 开关项的更多介绍，请参阅附录 A 中的表 A.18。

示例 12.4

```
1    $ perl -V  (Windows Command Line)
     Summary of my perl5 (revision 5 version 8 subversion 8)
     configuration:
   Platform:
     osname=MSWin32, osvers=4.0, archname=MSWin32-x86-multi-thread
     uname=''
     config_args='undef'
        hint=recommended, useposix=true, d_sigaction=undef
      usethreads=define use5005threads=undef useithreads=define
      usemultiplicity=define
      useperlio=define d_sfio=undef uselargefiles=define usesocks=undef
      use64bitint=undef use64bitall=undef uselongdouble=undef
      usemymalloc=n, bincompat5005=undef
    Compiler:
      cc='cl', ccflags ='-nologo -GF -W3 -MD -Zi -DNDEBUG -O1 -DWIN32
          -D_CONSOLE -DNO_STRICT -DHAVE_DES_FCRYPT -DNO_HASH_SEED
          -DUSE_SITECUSTOMIZE -DPERL_IMPLICIT_CONTEXT -DPERL_IMPLICIT_SYS
          -DUSE_PERLIO -DPERL_MSVCRT_READFIX',
      optimize='-MD -Zi -DNDEBUG -O1',
        ... output continues here
2    @INC:

------------------------------------------------

3    $ perl -e 'print "@INC\n"'     (Windows)
      @INC:
      C:/perl/site/lib
      C:/perl/lib
      .

----------------------------------------------------------

4   $ perl -V  (Linux)
      @INC:
      /usr/local/lib/perl5/5.8.6/i686-linux
      /usr/local/lib/perl5/5.8.6
      /usr/local/lib/perl5/site_perl/5.8.6/i686-linux
      /usr/local/lib/perl5/site_perl/5.8.6
      /usr/local/lib/perl5/site_perl/5.8.5/i686-linux
      /usr/local/lib/perl5/site_perl/5.8.5
      /usr/local/lib/perl5/site_perl
       .
```

解释

1. Perl 命令的 -V 选项能够显示版本、配置和库信息。
2. 数组 @INC 中含有到库例程所在路径。
3. 在 Windows 命令行提示符下，打印数组 @INC 的内容。这些路径都通过逗号隔开，其第一个元素是 C:/Perl/site/lib，该路径指向 Windows 专用库函数的存放位置；第二个元素是 C:/Perl/lib，指向标准 Perl 库的位置。最后一个点号非常重要，表明当 Perl 搜索库文件时，会把当前工作目录（文件夹）下的所有库文件导入进来。
4. 含有“site”一词的路径名是随着站点不同而变化的，该路径中含有针对本站点架构而下载得到的库例程。其中有些特定的库文件已在标准发布包中提供。那些含有 site 或 linux（如果使用 Solaris UNIX，则也可能是 solaris）的路径分别指向针对 Linux 的库例程和模块。路径 /usr/local/lib/Perl5/5.8.6 指向标准 Perl 库的位置，而最后的点号则表示当前的工作目录。

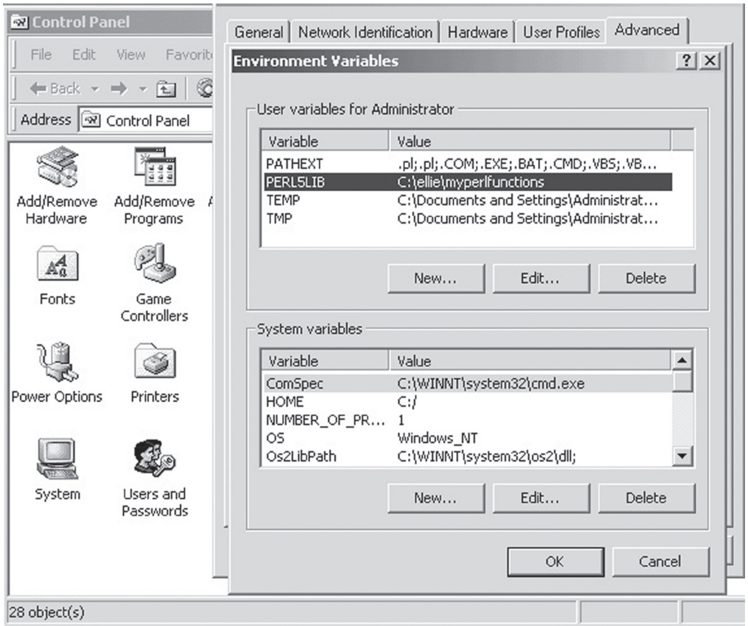


图 12-4　在 Windows 中设置环境变量 PERL5LIB

设置 PERL5LIB 环境变量。如果用户使用的是 UNIX/Linux 操作系统，并需要为数组 @INC 添加新路径元素的话，不妨在初始化文件中设定 PERL5LIB 环境变量。

在 C shell 或 TC shell 的 .login 文件中键入：

```
setenv PERL5LIB "directory path"
```

在 Bourne、Korn 及 Bash shell 中键入：

```
PERL5LIB = "directory path"; export PERL5LIB
```

如果使用的是 Windows，则可进入开始菜单，然后依次选择属性、系统属性、环境变量，最后点击新建按钮。参见图 12-4。

如需让自己设置的库例程优先于 @INC 数组列出的例程，可将如下代码行放在程序中：

```
unshift (@INC , " . ") ;
```

unshift 会把“.”加到数组 @INC 的最前面，使得当前工作目录路径成为搜索路径中的第一个元素。如果库文件位于不同目录的话，则应使用其全路径名，而不是点号。

12.2.2　包和 .pl 文件

标准 Perl 库提供的大多数 .pl 库例程都是在 Perl 4 时代编写的，它们由包含在包里面的子例程构成。这些库例程现在仍然可用，但其中许多库例程已由新的模块或 .pm 文件替代。

require 函数。 为了导入并执行标准 Perl 库例程或其他文件中的代码，应当使用 require 函数。它类似于 C 语言中的 #include 语句。require 函数能检查某个库文件是否已被导入。它和 eval 或 do

函数不同，后者是较为陈旧的文件导入方法。如果没有提供参数，该函数就默认导入 $_ 变量值。如果在 @INC 数组中没有正确的库路径，则 require 函数将执行失败，并返回如下出错消息：

Can't locate pwd.pl in @INC at package line 3.

require 函数在执行时会把文件加载到程序中，并更新 @INC 数组内容。

格式

```
require (Expr)
require Expr
require
```

载入标准库例程。 为提供一个有关 Perl 标准库例程的示例，示例 12.5 显示 pwd.pl 例程（该例程由 Larry Wall 编写）。pwd 包由两个子例程构成，负责更新 PWD 环境变量。其中，$ENV{PWD} 的值是当前工作目录。

请注意，在下列库例程中，包都声明在子例程定义之前，因此可把文件的其余部分放入 pwd 包中。所有的变量都输入该包。如需在 main 中调用它们，则应显式地将子例程的名字切换到 main 包中去，但其变量仍然是 pwd 包的本地变量（这里使用撇号代替双冒号来切换命名空间。撇号是 Perl4 中负责切换命名空间的符号）。

下面是一段来自 Perl 标准库中的 .pl 例程示例。

示例 12.5

```
(The pwd package)
    #
1   # Usage:
2   #    require "pwd.pl";
    #    &initpwd;
    #    ...
    #    &chdir($newdir);
3   package pwd;
4   sub main'initpwd {
        if ($ENV{'PWD'}) {
        local($dd,$di) = stat('.');
        local($pd,$pi) = stat($ENV{'PWD'});
        return if $di == $pi && $dd == $pd;
    }
    chop($ENV{'PWD'} = 'pwd');
    }
5   sub main'chdir {
        local($newdir) = shift;
        if (chdir $newdir) {
           if ($newdir =~ m#^/#) {
               $ENV{'PWD'} = $newdir;
           }
           else {
               local(@curdir) = split(m#/#,$ENV{'PWD'});
               @curdir = '' unless @curdir;
               foreach $component (split(m#/#, $newdir)) {
                   next if $component eq '.';
                   pop(@curdir),next if $component eq '..';
```

```
                }
                $ENV{'PWD'} = join('/',@curdir) || '/';
            }
        }
        else {
            0;
        }           # Return value
    }

6   1;    <---IMPORTANT
```

解释

1. 用法（usage）提示信息提示用户该如何使用这个包。
2. 在用法消息中，告诉用户应当对文件 pwd.pl 执行 require 操作。要调用的两个子例程是 inipwd 和 chdir。函数 chdir 接收一个参数，即目录名。
3. 声明包 pwd。这是文件中惟一的包。
4. 定义子例程 initpwd。注意，其名称限定为 main 包中的符号。这就意味着可以从 main 包中直接调用 initpwd，而不必提到包 pwd。
5. 定义子例程 chdir。
6. 对于 require 函数，在该包末尾要求加 1。如果该文件最后一个表达式的值为真，则 require 函数将不把文件加载到程序中。

下面这个示例在脚本中用到了库例程。

示例 12.6

```
(The Script)
    #!/bin/perl
1   require "ctime.pl";
2   require "pwd.pl";
3   &initpwd;          # Call the subroutine
4   printf "The present working directory is %s\n", $ENV{PWD};
5   &chdir ("../..");
6   printf "The present working directory is %s\n", $ENV{PWD};
7   $today=&ctime(time);
8   print "$today";

(Output)
4   The present working directory is /home/jody/ellie/perl
6   The present working directory is /home/jody
8   Wed  Mar 14 11:51:59 2007
```

解释

1. 这里加载了 Perl 标准库例程 ctime.pl。
2. 加载 Perl 标准库例程 pwd.pl。
3. 为 pwd.pl 函数调用子例程 initpwd，初始化 PWD 的值。
4. 打印环境变量 PWD 的当前值。
5. 调用 chdir 函数，改变当前工作目录。
6. 打印环境变量 PWD 更新后的值。
7. 通过 ctime.pl 中的子例程 ctime，将当天日期内容设定为易于阅读的格式。
8. 以新格式打印当天日期。

在 Perl 中加载自己的库。 下面的示例将展示如何创建自己的库例程，并通过 require 函数将它们载入到 Perl 脚本中。在加载用户自定义例程或者向库添加例程时，请读者确保例程的最后一行内容是 1;（非零值）。如果库文件的最后一行返回的不是一个真值，将从 require 函数得到如下报错：

average.pl did not return a true value at user.plx lines 3.

示例 12.7

```
(The midterms Script)
    #!/bin/perl
    # Program name: midterms
    # This program will call a subroutine from another file
1   unshift(@INC, "/home/jody/ellie/perl/mylib");
2   require "average.pl";
    print "Enter your midterm scores.\n";
    @scores=split(' ', <STDIN>);
3   printf "The average is %.1f.\n", average::ave(@scores);
    # The ave subroutine is found in a file called average.pl
------------------------------------------------------------
4   $ cd mylib          # Directory where library is located
------------------------------------------------------------

(The Script)
5   $ cat average.pl    # File where subroutine is defined
6   package average; # Declare a package
     # Average a list of grades
7   sub ave {
8       my(@grades)=@_;
        my($num_of_grades)=$#grades + 1;
        foreach $grade ( @grades ){
             $total += $grade;
        }
9       $total/$num_of_grades;   # What gets returned
    }
10  1; # Make sure the file returns true or require will not succeed!
```

解释

1. unshift 函数负责将指向个人目录 mylib 的路径名添加到 @INC 数组中。
2. require 函数首先检查 @INC 数组内容，获得要在其中查找 .pl 文件的所有目录清单。require 函数将加载 Perl 函数 average.pl。
3. 调用 ave 函数，并将返回值保存到标量 $average 中。由于 ave 函数定义在库文件中一个名叫 average 的包内，因此在调用 ave() 时必须在它前面加上包名（和两个冒号）。否则，Perl 将尝试在当前包 main 中查找 ave()。
4. 在命令行将目录切换到 mylib。美元符号在这里是 shell 提示符。
5. 现在查看文件 average.pl 的内容。
6. 定义一个名为 average 的包。
7. 定义子例程 ave()。
8. 该子例程以一组分数值作为参数。通过 my 函数将这些参数变量本地化。
9. 求表达式值并返回之。
10. 本语句的值为 true，并位于文件末尾。在加载文件时，require 函数要求返回值为 true。

12.2.3 模块和 .pm 文件

在使用标准 Perl 库提供的模块（即以 .pm 为扩展名的文件）时，读者必须先要保证 @INC 数组含有指向发布包库的全路径，并使用 use 函数引用所需模块名。

如需了解特定库模块的功能，可使用 perldoc 命令获得帮助文档（perldoc 命令对于库中 .pl 文件是无效的）。例如，如需了解 CGI.pm 模块，则可在命令行中输入：

```
perldoc CGI
```

就能显示 CGI.pm 模块相关的文档。如果输入：

```
perldoc English
```

则能显示 English.pm 模块相关文档。

use 函数（模块和编译指示）。use 函数用于在编译时将 Perl 模块和编译指示符载入到程序中。如果模块的文件名不含 .pm 扩展名的话，use 函数就不会载入该模块。require 函数也能完成同样的工作，但它不能在运行时导入（import）和加载模块。

模块是库里面的文件，其行为取决于所在的环境。标准 Perl 库中的模块拥有 .pm 扩展名，用户亦可在子目录中找到它们。例如，模块 Bigfloat.pm 在子目录 Math 中。如需使用位于其他子目录中的模块，则应当在目录名后面跟上两个冒号，然后给出模块名称，譬如 Math::Bigfloat.pm。（尽管这两个冒号使 Bigfloat.pm 模块看上去像是位于一个 Math 包里面，但在这里的上下文中，这两个冒号只负责隔开 Math 目录 / 文件夹和模块名。在 UNIX 型操作系统中，Perl 将把这对冒号替换为正斜杠；而在 Windows 系统中，则替换为反斜杠。）

编译指示（pragma）告诉编译器要求程序行为必须满足某种形式，否则就退出程序。常见的编译指示符包括 lib、strict、sub 和 diagnostics。有关模块和编译指示符的详细列表请参阅附录 A 中的表 A.4 和表 A.5。

在面向对象的术语中，子例程又称作模块。如果读者在这里看到涉及方法（method）的诊断信息，则可以认为是子例程。库中很多模块都使用了面向对象的 Perl。但本章讨论的模块不要求用户了解对象的概念。有关如何使用面向对象模块的内容，请参阅第 14 章“面向对象的 Perl”）。

格式

```
use Module;
use Module ( list );
use Directory::Module;
use pragma (list);
no pragma;
```

12.2.4 导出和导入

在进出口业务中，有的人要出口货物，而另一些人则要进口货物。假定某位加州酿酒商的酒窖里存有四种极品葡萄酒，他决定向客户销售其中的三种，而把最好的那一种留给自己享用。为此他打印了一份出售清单，并将它钉在了酒窖的墙上，购买者只能从这份清单里选择买哪种酒。购买者负责提供输入。当然，输入者可以要求购买所有的四种酒，但是如果按照出售清单做的话，就只能选择列在清单上的那几种。

在使用 Perl 模块时，用户就像上面的购买者，能从模块提供的导出（export）列表中导入（import）所需符号（子例程和变量）。用户可从其他模块中获得符号，并将它们添加到自己的符号

表里。此时，用户可以取列表上的默认项，也可导入列表中的其他具体项，甚至还可屏蔽列表中的其他所有符号。导入和导出只是一种命名空间导入方式，负责将符号导入程序包中，而不必使用模块的包名和双冒号来完全限定所有导入项的名称，如 &Module::fun1。导入的内容可以列在 use 指令后面，譬如 use Module qw(fun1 fun2)。

Exporter 模块。 导出模块操作能把模块中的符号发送给使用模块的用户。标准 Perl 库中的模块 Exporter.pm 为导出模块中的变量或子例程提供了必要的符号支持。它实现了一种导入方法，允许模块把函数和变量导出到用户命名空间。如前所述，符号是以一个 typeglob 散列的形式保存在包内符号表上的。导入例程负责为符号创建一个别名，进而将该符号从一个包里导出，并用到另一个包中去。考虑如下语句：

```
*Package_mine::somefunction = \&Package_exporter::somefunction
```

上述名为“Package_mine”的包将从“Package_exporter”中导入一个叫做“somefunction”的符号。该符号是子例程“somefunction”的程序名。在此过程中，程序会把一个指向原包内“somefunction”的引用赋值给“Package_mine”中的 typeglob 结构，譬如“Package_mine”的符号表。

Exporter 模块也实现了同前面所述机制类似的一种导入方法。尽管可以定制属于自己的导入方法，但大多数模块还是使用了 Exporter 模块，因为后者的接口在扩展性和易用性方面都很出色。

在针对某个模块处理 use 语句时，Perl 会自动调用其导入方法。用户可在 perlfunc 和 perlmod 中找到有关模块和 use 语句的说明文档。如要深入理解 Exporter，首先必须了解模块的概念，以及 use 语句的工作原理。Exporter.pm 是面向对象的模块，其功能和类（class）相似。其他模块只能从 Exporter.pm 类中继承符号导出能力（有关面向对象的知识请参阅第 14 章）。所有继承类必须在数组 @ISA 中予以列举。

```
require Exporter;
our @ISA = qw (Exporter); ⊖
```

在默认情况下，@EXPORT 数组中列出的名称将会切换到模块调用者的命名空间上。只有在显式请求时，才会将 @EXPORT_OK 数组中的名称添加到用户所在命名空间上。而 @EXPORT_FAIL 数组则列出了不能导出的符号。如果使用 use 语句导入模块，并在模块名上添加了括号的话（譬如 use Module()），则不会将符号导出到模块中。表 12-1 介绍了导出模块以及用到它的模块情况。

表 12-1 导出符号

导出模块	含　义
package Testmodule;	包声明
require Exporter;	使用 Exporter.pm 模块将符号从包导出到另一个包
our @ISA = qw(Exporter);	@ISA 含有在导出执行时需要的基本类名
our @EXPORT=qw($x @y z);	列表中的符号自动导出到该模块的调用方
our @EXPORT_OK=qw(fun b c);	只有收到请求时，才由模块调用者导出列表中的符号

⊖ 注意，当用作 require 的参数时，Exporter 模块并不位于双引号中，并且没有出现 .pm 扩展名。这就告诉 Perl 编译器两件事：如果需要将模块 Math::BigFloat 转换为 Math/BigFloat，或者在模块中有间接的方法引用的话，则将它们当作面向对象的方法来对待，而不是普通的子例程。

（续）

导出模块	含　义
our @EXPORT_FAIL=qw(fun3 e);	不导出的符号[a]
our %EXPORT_TAGS={ *':group1'=> [qw (a b c)] ,* *':group2'=> [qw ($x @y %c)] ,* *);*	键 'group1' 统一代表符号 a、b 和 c（函数名）；'group2' 则统一代表符号 $x、@y 和 %c
导入模块	**含　义**
use Testmodule;	加载 Testmodule
use Testmodule qw(fun2);	加载 Testmodule，导入 fun2
use Testmodule();	加载 Testmodule，不导入任何符号
use Testmodule qw(:group1 !:group2);	Testmodule 从 group1（见上面的 %EXPORT_TAGS 散列值）导入符号，同时不导入 group2 中的符号
use Testmodule qw(:group1 !fun2);	Testmodule 从 group1 中导入符号，同时不导入 fun2
use Testmodule qw(/^fu/)	Testmodule 导入所有名字以 fu 开头的符号
SomeModule.pm	**模块调用代码**
package SomeModule.pm *use Exporter;* *our @ISA = qw(Exporter);* *our @EXPORT = qw(a b c);* *sub a{}* *sub b{}* *sub c{}* *sub d{}* *1;*	use SomeModule; &a ; &b ; &c ; # 自动从 @EXPORT 数组中导入相应函数和变量 &SomeModule::d #d 不在列表 @EXPORT 中 # 因此必须定义其全路径
package SomeModule.pm *use Exporter;* *our @ISA = qw(Exporter);* *our @EXPORT-OK = qw(a b c);* *sub a{}* *sub b{}* *sub c{}* *sub d{}* *1;*	use SomeModule qw(a c); # 必须要求输入符号 &a ; # 否则就不会导入它们 &c ; &SomeModule::c # 必须给出符号的全名 SomeModule::d # 不从 @EXPORT_OK 列表请求符号

a. 变量名前带有特殊符号；而子例程不含特殊符号。a、b、c 指的是具有这些名字的子例程。

使用 perldoc 获取 Perl 模块文档。当准备使用 Perl 模块时，可以通过内建的 perldoc 命令检索 Perl5 模块相关文档，这些文档都通过特殊的 pod 指示符格式化（有关此类格式的详细信息，请参阅第 15 章）。下面这个示例即来自于 CGI.pm 文档。

请注意：读者若想实际运行模块 CGI.pm 所示代码的话，请剪切和粘贴下面高亮部分的内容，并置入另一个文件中。然后把这个文件保存到 Web 服务器根目录下的 cgi-bin 子目录。如果使用 UNIX 的话，还应当借助 chmod 命令打开文件的执行权限。接着便可在浏览器中执行该脚本了。读者如果还不清楚如何执行 CGI 程序的话，请参阅本书第 16 章，了解相关指令的详情。

示例 12.8

```
(At the Command Line)
1  perldoc CGI
(Output)
NAME
    CGI - Simple Common Gateway Interface Class

SYNOPSIS
      # CGI script that creates a fill-out form
      # and echoes back its values.

2     use CGI qw/:standard/;
3     print header,
           start_html('A Simple Example'),
           h1('A Simple Example'),
           start_form,
           "What's your name? ",textfield('name'),p,
           "What's the combination?", p,
           checkbox_group(-name=>'words',
                          -values=>['eenie','meenie','minie','moe'],
                          -defaults=>['eenie','minie']), p,
           "What's your favorite color? ",
                popup_menu(-name=>'color',
                -values=>['red','green','blue','chartreuse']),p,
          submit,
           end_form,
           hr;

      if (param()) {
          my $name      = param('name');
          my $keywords  = join ', ',param('words');
          my $color     = param('color');
          print "Your name is",em(escapeHTML($name)),p,
                "The keywords are: ",em(escapeHTML($keywords)),p,
                "Your favorite color is ",em(escapeHTML($color)),
                hr;
      }

ABSTRACT
    This perl library uses perl5 objects to make it easy to create Web
    fill-out forms and parse their contents. This package defines CGI
-- More  --
```

解释

1. perldoc 命令能为那些使用 Perl POD 指示符的模块生成帮助文档。（有关如何使用纯 Ole 文

档（即 Plain Ole Documentation，POD）的细节请参阅第 15 章）。

2. 本行负责以基于函数的形式加载 CGI 模块，并导入所有标准符号，从而创建一段含有 HTML 和 Perl 函数的 CGI 脚本。CGI.pm 模块自身通过 %EXPORT_TAGS 散列表定义了一组符号列表，其中“standard”是键，后面跟有一组相应值（以及含有大量函数名的一组标签）：':standard' => [qw/:html2 :html3: html4: form: cgi/]。这些符号将自动导入到调用该模块的代码中。
3. 这部分文档告诉用户如何使用 CGI.pm 中的某些特性。有关 CGI 的更多信息，请阅读第 16 章。

使用标准 Perl 库中的 Perl5 模块。下面的 English.pm 模块为内建变量（如 $_ 和 $/）定义了一些别名。对于那些也属于 awk 编程语言一部分的变量，其变量同时具有长短两种形式。例如，存储当前记录数的变量在 Perl 中表示为 $.，而在 awk 中则表示为 NR。其英文名分别是 $RS(awk) 或 $INPUT_RECORD_SEPARATOR(Perl）。

示例 12.9

```
(The Script)
    #!/usr/bin/perl
1   use English;     # Use English words to replace
                     # special Perl variables
2   print "The pid is $PROCESS_ID.\n";
3   print "The pid is $PID.\n";
4   print "The real uid $REAL_USER_ID.\n";
5   print "This version of perl is $PERL_VERSION.\n";

(Output)
2   The pid is 948.
3   The pid is 948.
4   The real uid 9496.
5   5.6.0.
```

解释

1. 在编译时，使用 use 指令加载 English.pm 模块。
2. 打印该进程的 id 号。
3. 变量 $PID 的内容和 #PROCESS.ID 值相同。
4. 打印该程序用户的真实用户 id。
5. Perl 版本为 5.6.0。

下面是取自 Perl 标准库中的一个 .pm 文件示例。

示例 12.10

```
(A Module from the Standard Perl Library)[a]
1   package Carp;
    # This package implements handy routines
    # for modules that wish to throw
    # exceptions outside of the current package
2   require Exporter;
3   @ISA = Exporter;
4   @EXPORT = qw(confess croak carp);
5   sub longmess {
6       my $error = shift;
        my $mess = "";
```

```
        my $i = 2;
    my ($pack,$file,$line,$sub);
    while (($pack,$file,$line,$sub) = caller($i++)) {
        $mess .= "\t$sub " if $error eq "called";
        $mess .= "$error at $file line $line\n";
        $error = "called";
    }
    $mess || $error;
    }
    sub shortmess {
        my $error = shift;
        my ($curpack) = caller(1);
        my $i = 2;
        my ($pack,$file,$line,$sub);
        while (($pack,$file,$line,$sub) = caller($i++)) {
           return "$error at $file line $line\n"
                        if $pack ne $curpack;
    }
    longmess $error;
    }
7   sub confess { die longmess @_; }
8   sub croak { die shortmess @_; }
9   sub carp { warn shortmess @_; }
```

解释

1. 这是一个包声明。程序将在包文件 Carp.pm 之后对该包进行命名。函数 carp、croak 和 confess 将生成出错信息，等效于 die 和 warn。其区别在于，当使用 carp 和 croak 时，会在调用例程中产生错误的那一行报错；而 confess 函数则会打印出相应堆栈回溯信息，从而说明产生错误的子例程链。它是在调用的那一行打印出错信息的。
2. 这里 require Exporter 模块，使得其子例程和变量能供其他程序使用。
3. 数组 @ISA 中含有该模块需要用到的包名。Perl 在 @ISA 数组中列出该包用到其他模块，并通过这种方式实现其继承性。
4. 数组 @EXPORT 列出了从该模块默认导出到模块调用者的子例程。在使用该模块的时候，不需其他设置就可以调用其子例程 confess、croak 以及 carp。由于 longmess 和 shortmess 都不在导出列表上，因此不能直接调用这两个子例程。
5. 这是模块的子例程定义。
6. 本行把出错信息作为参数传递给 confess 函数，并将其转存到标量 $error 中。
7. 将子例程 confess 定义为以 longmess 返回值作为参数调用 die 函数。
8. 将子例程 croak 定义为以 shortmess 返回值作为参数调用 die 函数。
9. 将子例程 carp 定义为以 shortmess 返回值作为参数调用 warn 函数。

12.2.5 如何“use”来自标准 Perl 库的模块

下面这个例子显示了该如何使用来自标准 Perl 库的 Carp 模块。使用模块的第一步是仔细阅读相关文档。读者可利用 perldoc 命令完成这一步。

```
$ perldoc Carp

NAME
```

```
        carp - warn of errors (from perspective of caller)

        cluck - warn of errors with stack backtrace (not exported by default)

        croak - die of errors (from perspective of caller)

        confess - die of errors with stack backtrace

        shortmess - return the message that carp and croak produce

        longmess - return the message that cluck and confess produce
    SYNOPSIS
        use Carp;
        croak "We're outta here!";

        use Carp qw(cluck);
        cluck "This is how we got here!";

        print FH Carp::shortmess("This will have caller's details added");
        print FH Carp::longmess("This will have stack backtrace added");

    DESCRIPTION
        The Carp routines are useful in your own modules because they act
        like die() or warn(), but with a message which is more likely to be
        useful to a user of your module. In the case of cluck, confess, and
        longmess that context is a summary of every call in the call-stack.
        For a shorter message you can use carp, croak or shortmess which
        report the error as being from where your module was called. There
        is no guarantee that that is where the error was, but it is a good
        educated guess.
             <continues here>
```

“use”指示符保证程序在编译时载入了所需的模块。如果在模块名后面跟有一个列表，则这个列表的内容代表了从该模块中导出的模块，以便供读者自己的程序调用（导入）。在 Carp.pm 模块中，有一个名为 croak 的函数。在调用时，用户可以不用 :: 结构指明符号的全名（Carp::croak），直接便可调用这个 croak 函数。

示例 12.11

```
(Using a Module from the Standard Perl Library in a Script)
    #!/bin/perl
1  use Carp qw(croak);

2  print "Give me a grade: ";
$grade =<STDIN>;
3  try($grade);       # Call subroutine

4  sub tyr{
5    my($number)=@_;
6    croak "Illegal value: "if $number < 0  ||  $number > 100 ;
   }

(Output)
2  Give me a grade: 200
6  Illegal value: at expire line 13
       main::try called at expire line 8
```

解释

1. 在当前包 main 中使用（载入）Carp 模块。由于 croak 函数位于模块名后面的列表中，因此它是惟一一个可在本脚本中不指定全名而直接调用的子例程。譬如，如果读者使用 confess 而不是 Carp::confess 的话，程序便会退出执行，并报一条出错信息：String found where operator expected at cluck.plx line9，near "confess" Illegal value : " "（Do you need to predeclare confess?) ...
2. 要求用户输入。
3. 调用子例程 try，并将标量 $grade 传递给它。
4. 定义 if 子例程 try。
5. 将传入的参数赋值给 $number。
6. 以出错信息为参数调用 croak 函数。croak 函数由 Carp 模块导出。如果 $number 变量值不在 0 到 100 之间的话，程序便会退出。出错信息将显示程序退出的行号，以及包名、子例程名和调用子例程的行号。

示例 12.12

```
(Using a Module from the Standard Perl Library in a Script)
   #!/bin/perl
1  use Carp qw(cluck);  # cluck not exported by default

  print "Give me a grade: ";
  $grade = <STDIN>;
2  try($grade);     # Call subroutine

   sub try{
      my($number)=@_;
      cluck "Illegal value: " if $number < 0 || $number > 100;
   }
  print "That was just a warning. Program continues here.\n";

(Output)
2   Give me a grade: 200
6   Give me a grade: 200
    Illegal value:  at cluck.plx line 9
        main::try('200\x{a}') called at cluck.plx line 5
    That was just a warning. Program continues here.
```

解释

1. 在当前包 main 中使用（载入）Carp 模块。本脚本中将使用 cluck 函数。如果用户仔细看一下有关 cluck 函数的文档，就会发现该符号在默认情况下并不会导出到用户的命名空间中。因此，本程序将显式地导出该函数。
2. 要求用户输入。
3. 调用子例程 try，并将标量 $grade 传递给它。
4. 定义 if 子例程 try。
5. 将传入的参数赋值给 $number。
6. 以出错信息为参数调用 cluck 函数。该函数来自 Carp 模块。如果 $number 变量值不在 0 到 100 之间的话，程序便会发出一条警告信息。如果用户试图使用其他任何 Carp 函数的话，程序就会退出执行。为了使用其他这些函数，用户必须显式地导入它们，或者使用它们的全名，如：Carp::confess。

12.2.6 使用 Perl 创建自己的模块

下面这个示例展示了如何在 separate.pm 文件中创建模块，并在其他程序中使用该模块。虽然这个模块看起来与其他包一样，但用户必须在其中额外地载入 Exporter 模块、@ISA 数组和 @EXPORT 数组，以使它表现得更像一个模块。模块与 .pl 文件的区别在于，前者能把具体的符号导出到其他程序，或者导入其他模块中的符号。如要了解为 CPAN 创建模块框架的知识，请阅读“使用 h2xs 工具为 CPAN 创建扩展和模块”一节。

示例 12.13

```
(The Me.pm Module)
1   package Me;
2   use strict;  use warnings;
3   require 5.6;       # Make sure we're a version of Perl no
                       # older than 5.6
4   require Exporter;  # Exporter.pm allows symbols to be imported
                       # by others
5   our @ISA=qw(Exporter); # ISA is a list of base packages needed
                           # by this module
6   our @EXPORT_OK=qw(hello goodbye );  # List of your subroutines
                                        # to export
7   sub hello { my($name)=shift;
        print "Hi there, $name.\n" };
8   sub goodbye { my($name)=shift;
        print "Good-bye $name.\n";}
9   sub do_nothing { print "Didn't print anything.
                            Not in EXPORT list\n";}
    1;

-------------------------------------------------------------
    #!/usr/bin/perl
    # Program name: main.perl
10  use lib ("/home/ellie/Modules");   # A pragma to update @INC.
11  use Me  qw(hello goodbye);         # Import package
12  &hello ("Daniel");
13  &goodbye ("Steve");
14  &do_nothing;        # This was not on the Export list
                        # in Me.pm so cannot be imported unless
                        # explicitly with &Me::do_nothing
(Output)
12  Hi there, Daniel.
13  Good-bye Steve.
14  Undefined subroutine &main::do_nothing
```

解释

1. 该文件名为 Me.pm。它含有与自身名字相同的包，只不过不包括其扩展名。这里声明了 Me 包。
2. strict 编译指示符禁止使用全局变量，warning 指示符则负责在适当的时候发出警告信息，譬如出现只用一次的变量、未定义值等等。
3. 使用 require 保证所用 Perl 版本不低于 5.6。如果是，则退出脚本。
4. Exporter 模块是一种特殊的 Perl 模块。它允许用户通过 use 函数从特定模块中导入子例程（方法）。
5. @ISA 数组列出了该包所用所有子例程（方法）的属主包。这是 Perl 实现模块间继承效果的途径（详见第 14 章“面向对象的 Perl”）。
6. @EXPORT 数组列出了所有可以默认导出的子例程（方法）。@EXPORT_OK 数组列出了用

户在 use 语句中可以导出的子例程（方法）。如果用户没有请求，就不会得到它们。此处导出列表中的子例程是 hello 和 goodbye，将在下面予以定义。

7. 定义子例程 hello。
8. 定义子例程 goodbye。
9. 定义子例程 do_nothing。请注意，该子例程并不在导出列表中，即它不在 @EXPORT_OK 数组内。如果某个方法不在导出列表中，Perl 便会在当前包 main 中搜寻它。
10. lib 编译指示符告诉编译器在编译时更新 @INC 数，即：BEGIN{require "/home/ellie/Module"; import Module;}
11. use 函数使得 Perl 把 Me.pm 模块导入到该包中。
12. 以单个参数调用子例程 hello。导出该子例程。
13. 以单个参数调用子例程 goodbye。导出该子例程。
14. 不导出该子例程，因为它不在 Me.pm 中的导出列表（@EXPORT）内。打印出来的出错消息表明，包 main 不能识别该子例程。只有显式地提供包名 &Me::goodbye，才可以在包中使用这个子例程。

12.3 来自 CPAN 的模块

CPAN（Comprehensive Perl Archive Network）是由几百个 Perl 模块一起组成的中央库。如需查找最新的 CPAN 镜像，请访问 http://www.Perl.com/CPAN。（在第 16 章，本书将介绍如何从 CPAN 下载数据库模块，以及如何在程序中使用这些模块。）

相互依赖的 Perl 模块通过名字、作业和类别分门别类联系在了一起。用户可以在 CPAN module

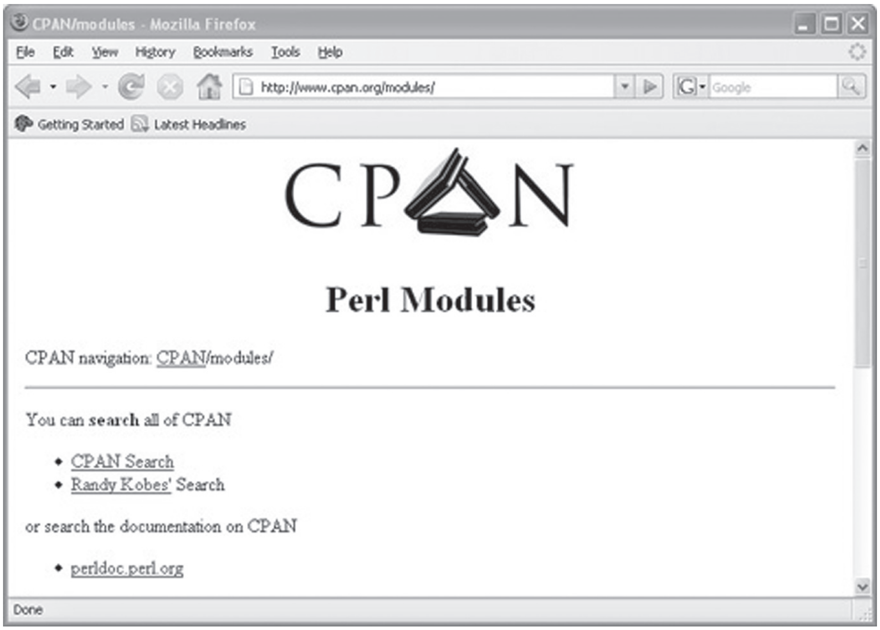


图 12-5 CPAN Web 站点上的 Perl 模块页面

目录下面找到所需模块，也可通过使用 http://www.Perl.com 中的 CPAN 搜索引擎找到它们。如需安装这些模块，CPAN 文档能为用户提供简单易懂的安装指南。下面这个 Web 页面显示了信息是如何分类的，并给出了部分模块列表。

ActivePerl 适用于 Linux、Solaris 和 Windows 操作系统，它含有 Perl Package Manager（用于安装 CPAN 模块包）以及完整的在线帮助。PPM 允许用户访问模块库、安装新模块、以及相对方便地升级老版本包。

如需使用 ActiveState 软件包库，用户可访问 www.activestate.com/ppm packages/5.6。

Cpan.pm 模块。 Cpan.pm 模块用于从 CPAN 站点查询、下载和构建 Perl 模块。它提供了交互式和批处理两种使用模式。其设计目标是支持 Perl 模块及其扩展组件的自动化安装。这些模块可以来自一个或者多个 CPAN 镜像站点，并解压到指定的目录中。若要学习有关该模块的更多内容，请在系统命令行环境中输入：

```
$ perldoc Cpan
```

读者将能看到如下内容：

```
NAME
    CPAN - query, download and build perl modules from CPAN sites

SYNOPSIS

   Interactive mode:

      perl -MCPAN -e shell;

   Batch mode:

      use CPAN;

      autobundle, clean, install, make, recompile, test

DESCRIPTION
    The CPAN module is designed to automate the make and install of perl
    modules and extensions. It includes some searching capabilities and
    knows how to use Net::FTP or LWP (or lynx or an external ftp client) to
    fetch the raw data from the net.

    Modules are fetched from one or more of the mirrored CPAN
    (Comprehensive Perl Archive Network) sites and unpacked in a
    dedicated directory.

    The CPAN module also supports the concept of named and versioned
    *bundles* of modules. Bundles simplify the handling of sets of related
    modules. See Bundles below.

    The package contains a session manager and a cache manager. There is no
    status retained between sessions. The session manager keeps track of
    what has been fetched, built and installed in the current session. The
    cache manager keeps track of the disk space occupied by the make
    processes and deletes excess space according to a simple FIFO
    mechanism.

    For extended searching capabilities there's a plugin for CPAN available,
```

```
the CPAN::WAIT manpage. 'CPAN::WAIT' is a full-text search engine that
indexes all documents available in CPAN authors directories. If
'CPAN::WAIT' is installed on your system, the interactive shell of
<CPAN.pm> will enable the 'wq', 'wr', 'wd', 'wl', and 'wh' commands
which send queries to the WAIT server that has been configured for your
installation.
                      <continues>
```

示例 12.14

```
1   $ h2xs -A -n Exten.dir
    Writing Exten.dir/Exten.dir.pm
    Writing Exten.dir/Exten.dir.xs
    Writing Exten.dir/Makefile.PL
    Writing Exten.dir/test.pl
    Writing Exten.dir/Changes
    Writing Exten.dir/MANIFEST

2   $ cd Exten.dir

3   $ ls

4   $ more Exten.dir.pm
    package Exten.dir;

    require 5.005_62;
    use strict;
    use warnings;

    require Exporter;
    require DynaLoader;

    our @ISA = qw(Exporter DynaLoader);

    # Items to export into callers namespace by default.
    # Note: do not export names by default without
    # a very good reason. Use EXPORT_OK instead.
    # Do not simply export all your public
    # functions/methods/constants.

    # This allows declaration use Exten.dir ':all';
    # If you do not need this, moving things directly
    # into @EXPORT or @EXPORT_OK will save memory.
perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.7602)
ReadLine support enabled

cpan> h

Display Information
 command  argument          description
 a,b,d,m  WORD or /REGEXP/  about authors, bundles, distributions, modules
 i        WORD or /REGEXP/  about anything of above
 r        NONE              reinstall recommendations
 ls       AUTHOR            about files in the author's directory

Download, Test, Make, Install...
 get                        download
 make                       make (implies get)
 test      MODULES,         make test (implies make)
 install   DISTS, BUNDLES   make install (implies test)
```

```
  clean                          make clean
  look                           open subshell in these dists' directories
  readme                         display these dists' README files

 Other
  h,?            display this menu        ! perl-code    eval a perl command
  o conf [opt]   set and query options    q              quit the cpan shell
  reload cpan    load CPAN.pm again       reload index   load newer indices
  autobundle     Snapshot                 force cmd      unconditionally do
  cmd

 cpan>
```

使用 PPM

PPM 是 ActivePerl（www.activestate.com）提供的一种程序管理器。它非常易于使用，并能运行在 Linux、Windows 和 Mac OS 等操作系统上。它既提供了 GUI 界面，也提供了命令行接口。在激活 PPM（Perl Package Manager）后，它会打开一个窗口，并列出所有当前已安装的模块。然后用户可以使用这个窗口搜索指定模块，或者安装、更新和移除模块。在日常维护工作中，上述使用模式要比 CPAN 更为方便。

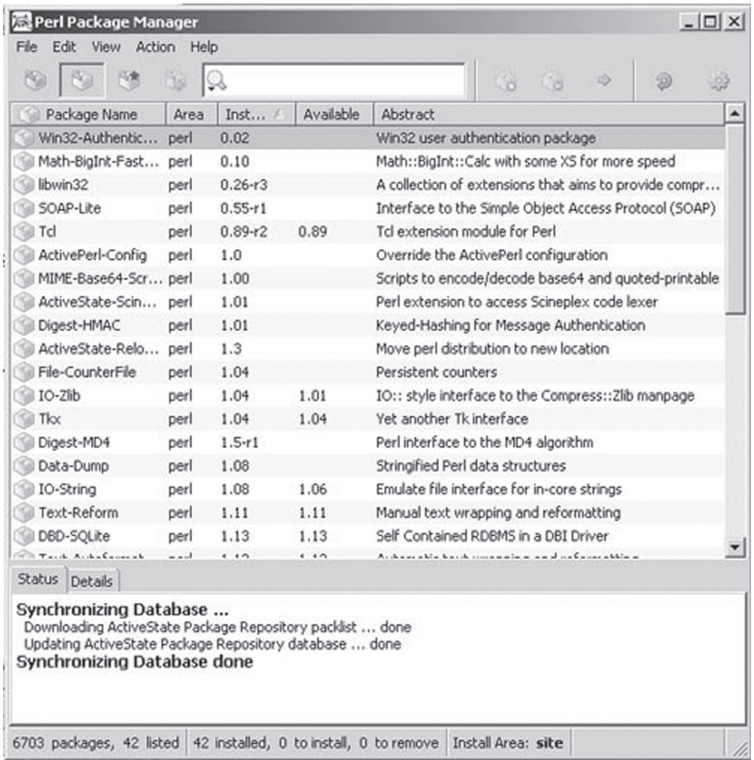


图 12-6 PPM GUI

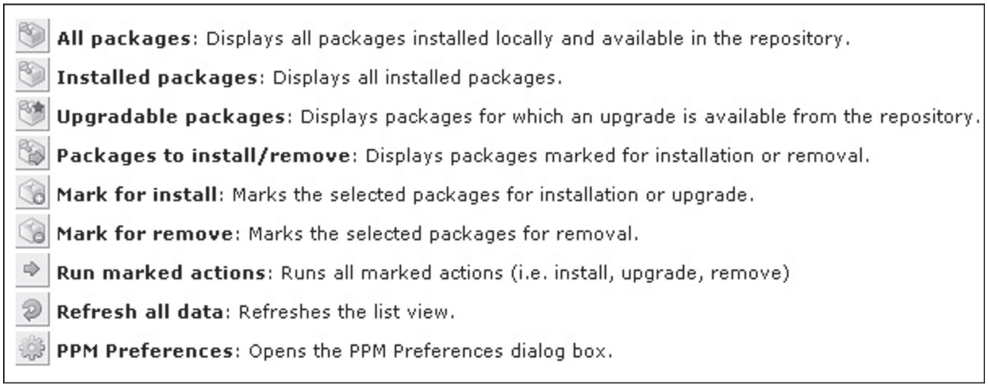


图 12-7 PPM 窗口内各个图标的含义

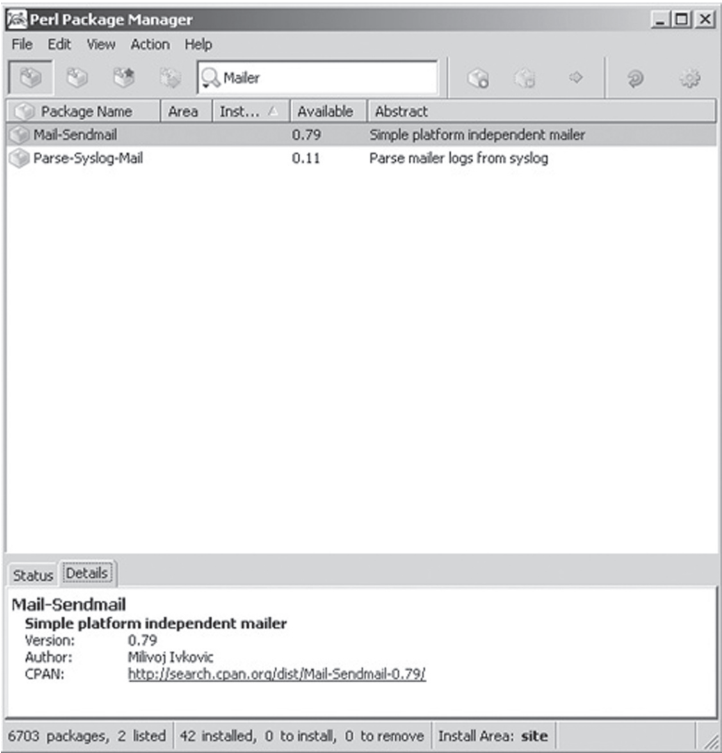


图 12-8 搜索一个名为 SendMail 的 Perl 模块

使用 h2xs 工具为 CPAN 创建扩展和模块。 h2xs 工具是 Perl 发布包中携带的标准应用程序。它负责创建目录和框架文件，常用于创建模块和添加 C 语言扩展时。如需获得有关该工具的完整描述信息，请在系统提示符下键入：

```
$perldoc h2xs
```

示例 12.15

```
1  $ h2xs -A -n Exten.dir
   Writing Exten.dir/Exten.dir.pm
   Writing Exten.dir/Exten.dir.xs
   Writing Exten.dir/Makefile.PL
   Writing Exten.dir/test.pl
   Writing Exten.dir/Changes
   Writing Exten.dir/MANIFEST

2  $ cd Exten.dir

3  $ ls

4  $ more Exten.dir.pm
   package Exten.dir;

   require 5.005_62;
   use strict;
   use warnings;

   require Exporter;
   require DynaLoader;

   our @ISA = qw(Exporter DynaLoader);

   # Items to export into callers namespace by default.
   # Note: do not export names by default without
   # a very good reason. Use EXPORT_OK instead.
   # Do not simply export all your public
   # functions/methods/constants.

   # This allows declaration use Exten.dir ':all';
   # If you do not need this, moving things directly
   # into @EXPORT or @EXPORT_OK will save memory.
   our %EXPORT_TAGS = ( 'all' => [ qw() ] );

   our @EXPORT_OK = ( @{ $EXPORT_TAGS{'all'} } );

   our @EXPORT = qw( );
   our $VERSION = '0.01';

   bootstrap Exten.dir $VERSION;

   # Preloaded methods go here.

   1;
   _ _END_ _
   # Below is stub documentation for your module.
   # You'd better edit it!

   =head1 NAME

   Exten.dir - Perl extension for blah blah blah

   =head1 SYNOPSIS
   use Exten.dir;
```

```
    blah blah blah

    =head1 DESCRIPTION

    Stub documentation for Exten.dir, created by h2xs. It looks like
the author of the extension was negligent enough to leave the stub
unedited.

    Blah blah blah.

    =head2 EXPORT

    None by default.

    =head1 AUTHOR

    A. U. Thor, a.u.thor@a.galaxy.far.far.away

    =head1 SEE ALSO

    perl(1).

    =cut
```

解释

1. h2xs 工具创建一个子目录(此处名叫 Exten.dir)。该子目录含有六个文件,用于创建符合 CPAN 格式的模块。在为 CPAN 创建模块之前,请先到 www.cpan.com/modules/00modlist.long.html 确认一下有没有其他人编写过该模块。这样做可以收到事半功倍的效果。
2. 切换到 h2xs 创建的新目录中。
3. 这是 h2xs 工具（未曾显示）所创建文件的清单。MANIFEST 文件含有在这个目录中创建文件的列表,并列出了模块发布时需要携带的附加文件。Makefile.PL 文件负责生成 Makefile。Exten.dir.pm 是含有扩展的框架模块，而 Exten.dir.xs 则含有 XSUB 例程，负责加载 C 扩展。
4. 这是一个框架模块，能帮助用户正确地创建模块。

12.4 读者应当学到的知识

1. 什么是 Perl 程序里的默认包?
2. 什么是符号表（symbol table）?
3. 如何查看符号表?
4. 什么是 @INC 数组?
5. 什么是编译指示（pragma）?
6. 环境变量 PERL5LIB 是做什么用的?
7. 什么时候应当使用 require ?
8. require 和 use 的区别是什么?
9. 什么是 Exporter 模块?
10. 什么是 EXPORTER_OK 数组? 什么是 %EXPORTER_TAGS 散列?
11. 在 .pl 或 .pm 文件末尾放入一个“1”的涵义是什么?

12. 如何更新 @INC 数组？
13. 如何从另一个包中访问子例程？
14. 如何维护变量的私有性？
15. 从 CPAN 安装模块最简便的方法是什么？

12.5 下章简介

下一章将关注指针（pointer），又称引用（reference）。读者将学习如何借助指针创建复杂数据结构、如何创建匿名变量、如何获取对指针的引用内容、以及如何把指针传入子例程。

读者还将学到为什么要在 Perl 中使用指针。

练习 12 把所有 Perl 都放入包内

1. 编写脚本 myATM。它含有两个包：Checking 和 main。然后把该文件拆分为一个用户文件和一个模块文件。
2. 在 myATM 脚本中，声明一个名叫 Checking 的包。
   它含有一个名叫 balance 的 my 变量，并赋值为 0。
   一开始，它含有三个子例程：
   a. *get_balance*
   b. *deposit*
   c. *withdraw*
3. 在 main 包（位于同一文件内）中，创建一个 here 文档，负责生成如下输出内容：
   1) Deposit
   2) Withdraw
   3) Current Balance
   4) Exit
   然后要求用户选择其中一个菜单项。如果选择的不是第 4 项，程序便进入循环，然后重新显示菜单内容，并等待用户选择另一个事务。
   在 main 包中调用子例程。为此必须指定 Checking 包名和两个冒号。
   如果用户选择第 4 项，则在退出程序前，将当前日期和余额打印到文件 register。
   如果不调用 get_balance 子例程，还能打印余额值吗？
4. 重新编写 Checking 包，当存在 register 文件时，便从该文件中读取余额数值；否则就从余额为 0 开始。每当程序退出时，均把当前日期和余额写入到 register 文件中。
5 a. 在第 11 章的练习 11 中，读者曾经编写过一段名为 trigger 的程序，它含有一个子例程 mileage。该例程要求用户输入驾驶里程数和所耗油量。该例程将计算并返回用户的耗油率（即每加仑汽油行驶的里程）。传递给子例程的参数包括驾驶里程数和所耗油量。这两个参数都赋值给了相应的 my 变量。如果读者尚未编写过 trigger 程序，这里我们还可再来一次。
   b. 如果表示油量的输入值是一个非数值型量、负数或者 0 的话，就调用 croak 函数打印出错信息。

现在创建一个名叫 myfunctions 的子目录。切换到该目录,并创建一个名叫 mileage.pl 的文件。然后把子例程 mileage 放入该文件 mileage.pl 中。

c. 在 trigger 脚本中，更新 @INC 数组内容，并借助 require 函数导入 mileage 子例程（请确保子例程中位于终止花括号后的最后一行内容为 1)。

6. 将 Checking 包从 myATM 脚本移动到另一个名叫 Checking.pm 的文件中。然后把 Checking.pm 文件移入一个名叫“myModules”的目录内。在需要使用该模块的 myATM 脚本中更新 @INC 数组内容。在 myATM 脚本中使用 Checking 包。

第 13 章　这项工作需要引用吗

13.1　什么是引用，什么是指针

在 Perl 中，“引用（reference）”和“指针（pointer）”这两个词是可以互换的。一个引用是一种引用其他变量内容的变量。简而言之，它含有其他变量的地址信息。在向子例程传递参数时（详见第 11 章），我们已经见识了引用的威力。我们还可使用引用（或指针）去创建更为复杂的数据结构，例如一个键对应一组值的散列结构、或者元素为数组的数组等。此外，下一章在介绍 Perl 对象的创建时也会用到引用。

13.1.1　符号引用和硬引用

硬引用（hard reference）是一种标量型变量，其中含有其他类型数据的地址。它与 C 编程语言提供的指针类似[㊀]。本章将着重介绍硬引用。

一个位于符号表中的 Perl 变量只能持有一个指向变量值的硬引用。这种值可以像一个数字那么简单，也能像散列那样复杂。多个硬引用可以指向同一个变量值，但后者并不知道它们的存在。

一个符号引用（symbolic reference）能为另一个变量命名，而不是指向其变量值[㊁]。typeglobs 就是一种符号引用，其变量名前缀是一个 * 字符。它们不过是变量的一种别名而已。

读者可能还记得本书前面是如何使用 typeglobs 的。在第 11 章，我们已经讨论了早期的 Perl 是如何使用 typeglobs 来向子例程按引用传递参数的。而在第 12 章中，则使用 typeglobs 向某个包的符号表中导入符号。下面这条语句就用到了 typeglobs：

```
*town = *city;              # 任何 city 类型都能用 town 引用
```

上面的星号代表所有位于变量名前面的特殊字符，包括子例程、文件句柄和格式等。它能“glob”到符号表中含有这个变量名的所有符号上[㊂]。*town 是 *city 的别名。至于哪个符号引用哪个别名，则是由用户自己决定的。如需访问别名引用的实际值，只需在别名前面加上正确的特殊字符便可。例如：

㊀ 与 C 指针不同，Perl 指针是一种字符串，因此不能对它执行指针代数运算操作。

㊁ Wall,L., Programming Perl, O'Reilly & Associates: Sebastopol, CA, 1996, p. 244.

㊂ 这里与文件名替换（即 <p*>）时的 glob 操作不同。

```
Given: *town = *city
Then: $town refers to the scalar $city
      @town refers to the array @city
      $town{"mayor"} refers to an element of a hash $city{"mayor"}
```

示例 13.1 展示了另一种类型的符号引用。在这种符号引用中，一个变量值将指向另一个变量的名字。

示例 13.1

```
    #!/bin/perl
    # Program using symbolic references
1   $animal="dog";
2   $dog="Lady";
3   print "Your dog is called ${$animal}\n";# Symbolic reference
4   eval "\$$animal='Lassie';";
5   print "Why don't you call her ${$animal}?\n";

(Output)
3   Your dog is called Lady
5   Why don't you call her Lassie?
```

解释

1. 将标量 $animal 赋值为“dog”。Perl 会把变量名 animal 存入符号表中，并由一个引用指向其变量值 dog。
2. 将标量 $dog 赋值为字符串“Lady”。
3. 变量 ${$animal} 的值为 Lady。这是一个符号引用。其中 $animal 是一个变量，得到其值为 dog。第二个美元符号表示另一个变量 $dog，其变量值则是“Lady”。本行通过一个变量引用另一个变量。
4. eval 函数能对表达式求值，其效果就好像在某个独立的 Perl 小程序里一样。本行隐去了第一个美元符号。$animal 的值为 dog。文本美元符号位于求值结果前面，因此这条语句的内容就是 $dog="Lassie"。
5. 在 eval 之后，打印 ${$animal} 的值，即 $dog 的值“Lassie”。

strict 编译指示。为了避免在程序中不经意间误用符号引用，可使用 strict 编译指示，其参数为 refs。这样就能让 Perl 检查该符号引用是否用在程序中。在这里，我们通过 strict 编译指示再次执行前面的示例。

示例 13.2

```
    #!/bin/perl
    # Program using symbolic references
1   use strict "refs";
2   $animal="dog";
3   $dog="Lady";
4   print "Your dog is called ${$animal}\n";
5   eval "\$$animal='Lassie';";
6   print "Why don't you call her ${$animal}?\n";

(Output)
Can't use string ("dog") as a SCALAR ref while "strict refs" in use at symbolic.
plx line 4.
```

解释

1. strict 编译指示保证程序只使用硬引用，否则就在编译时退出程序，并打印如此脚本输出内容所示的出错信息。
3. 这是脚本的第 10 行。程序在此处中止执行，因为这里第一次出现了符号引用 ${$language}。
4. 这一行也含有符号引用，但程序将永远不会到达这里。因为 strict 编译指示已经在前面退出了该程序。

13.1.2 硬引用，指针

在第 11 章时，我们讨论过如何通过指针向子例程中传递引用。这里再回顾一下：硬引用是一种标量，它含有其他类型数据的地址。一个内容为地址值的变量也可称作指针，因为它指向了其他的地址或引用。这种类型的引用可以指向标量、数组、关联数组或者子例程。Perl 5 中引入的指针机制为用户提供了创建复杂数据类型的能力，譬如元素为数组的数组、元素为散列的数组、内容为散列的散列表等。所有原来使用 typeglobs 的示例，现在都可以选择用指针取而代之。指针提供了一种向子例程按引用传递参数的途径。

反斜杠运算符。反斜杠一元运算符可用于创建硬引用，类似于 C 语言中用于获取“地址”的 & 运算符。在下面这个示例中，$p 是一个引用。该引用赋值为标量 $x 的地址。

```
$p = \$x ;
```

下面是一个来自 Perl man 页面 perlref 部分的有关硬引用的示例：

```
$scalarref = \$foo;       # reference to scalar $foo
$arrayref  = \@ARGV;      # reference to array @ARGV
$hashref   = \%ENV;       # reference to hash %ENV
$coderef   = \&handler;   # reference to subroutine handler
$globref   = \*STDOUT;    # reference to typeglob STDOUT
$reftoref  = \$scalarref; # reference to another reference
                            (pointer to pointer, ugh)
```

按地址访问（dereference）指针。如果把引用（指针）的值打印出来，就可以看到其内容是地址。如果要切换到这个地址并获取存入其中的值（即按地址访问指针），用户必须在指针前面缀以两个“特殊”符号。第一个符号是一个美元字符，因为指针本身只是一种标量，所以前面必须带有代表它所指向数据类型的特殊符号。当使用更为复杂的类型时，用户还可使用箭头（中缀）运算符。

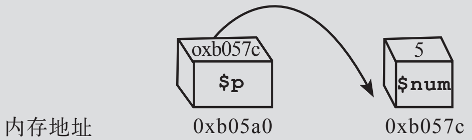
示例 13.3

```
(The Script)
    #!/bin/perl
    $num=5;
    $p = \$num;      # $p gets the address of $num
    print 'The address assigned $p is ', $p, "\n";
    print "The value stored at that address is $$p\n"; # dereference

(Output)
    The address assigned $p is SCALAR(0xb057c)
    The value stored at that address is 5
```

解释

1. 将标量 $num 赋值为 5。
2. 将标量 $p 赋值为 $num 的地址。这是一个反斜杠运算符函数。$p 可称作引用或指针。这两个术语是可以互换的。
3. 打印存储在 $p 中的地址。Perl 还表明其数据类型是 SCALAR。
4. 按地址访问指针 $p，为此在 $p 前面加上了另一个美元符号。这个美元符号告诉 Perl 去查看 $p 引用的变量值，即 $num。



示例 13.4

```
   #!/bin/perl
1  @toys = qw( Barbie Elmo Thomas Barney );
2  $num = @toys;
3  %games=("Nintendo"  => "Wii",
           "Sony"      => "PlayStation 3",
           "Microsoft" => "XBox 360",
         );
4  $ref1 = \$num;  # Create pointers
5  $ref2 = \@toys;
6  $ref3 = \%games;

7  print "There are $$ref1 toys.\n"; # dereference pointers
8  print "They are: ",join(",",@$ref2), ".\n";
9  print "Jessica's favorite toy is $ref2->[0].\n";
10 print "Willie's favorite toy is $ref2->[2].\n";

11 while(($key,$value)=each(%$ref3)){
      print "$key => $value\n";
   }
12 print "They waited in line for a $ref3->{'Nintendo'}\n";

(Output)
There are 4 toys.
They are: Barbie,Elmo,Thomas,Barney.
Jessica's favorite toy is Barbie.
Willie's favorite toy is Thomas.
Microsoft => XBox 360
Sony => PlayStation 3
Nintendo => Wii
They waited in line for a Wii
```

解释

1. 把一个列表赋值给数组 @toys。
2. 将数组 @toys 赋值给标量型变量 $num，从而返回数组元素总数。
3. 把键 / 值对赋值给散列 %games。
4. $ref1 是一个指针。借助反斜杠运算符把标量 $num 的地址赋值给它。

5. 将数组 @toys 的地址赋值给指针 $ref2。
6. 将散列 %games 的地址赋值给指针 $ref3。
7. 按地址访问指针，也就是说：转到 $ref1 指向的地址处，并打印这里保存的标量值。
8. 再一次按地址访问指针，也就是说，转到 $ref2 指向的地址处，获取并打印数组内容。
9. 箭头运算符 -> 用于按地址访问指针，并获取该数组的第一个元素。（也可写作 $$ref2[0]。）
10. 再次使用箭头运算符 -> 按地址访问指针，并获取该数组的第三个元素。
11. each 函数负责通过散列的指针检索其键和值。如需按地址访问散列，则应在指针变量前加上 % 符号。
12. 箭头运算符 -> 用于按地址访问指针，并获取其中键为“Nintendo”的散列值。（也可写作 $$ref3{" Nintendo "}。）

13.1.3　引用和匿名变量

在创建对变量的引用（指针）时，并不一定需要为该变量命名。如果某个变量或子例程没有名字，那么就称它是匿名（anonymous）的。如果把一个匿名变量（或子例程）赋值给标量，则这个标量就是对该变量（子例程）的引用。

箭头运算符（->）又称为中缀运算符，它可用于按地址访问指向匿名数组和散列的引用。尽管箭头运算符是可选的，但它能让程序更为易读。

匿名数组。匿名数组的元素位于方括号（[]）中。请读者不要把这些方括号与用到下标上的方括号混为一谈。这里的方括号用作给标量赋值的表达式。如果位于引号中，程序将不解释这类方括号。用户可利用箭头运算符来获取数组的单个元素值。

示例 13.5

```
(The Script)
    #!/bin/perl
1   my $arrayref = [ 'Woody', 'Buzz', 'Bo', 'Mr. Potato Head' ];
2   print "The value of the reference, \$arrayref is ",
                                    $arrayref, "\n";
    # All of these examples dereference $arrayref
3   print "$arrayref->[3]", "\n";
4   print $$arrayref[3], "\n";
5   print ${$arrayref}[3], "\n";
6   print "@{$arrayref}", "\n";

(Output)
2   The value of the reference, $arrayref is ARRAY(0x8a6f134)
3   Mr. Potato Head
4   Mr. Potato Head
5   Mr. Potato Head
6   Woody Buzz Bo Mr. Potato Head
```

解释

1. 把匿名数组元素赋值给数组引用 $arrayref。
2. 数组引用中含有匿名数组的数据类型和十六进制地址值。
3. 打印数组的第 4 个元素。指针变量 $arrayref 后面是指向待检索下标值的箭头运算符。
4. 这里并不需要真正的箭头运算符。第 4 行和第 5 行的方法可以访问元素。
6. 按地址访问指针，然后打印整个数组。花括号在这里是必需的。

匿名散列。匿名散列可通过花括号（{}）创建。用户可以混合使用数组和散列，从而创建复杂的数据类型。这些花括号不同于用在散列下标上的花括号。用户可以把匿名散列赋值给标量引用。

示例 13.6

```
(The Script)
    #!/bin/perl
1   my $hashref = { "Name"=>"Woody",
                    "Type"=>"Cowboy"
                  };
2   print $hashref->{"Name"}, "\n\n";
3   print keys %$hashref, "\n";
4   print values %$hashref, "\n";

(Output)
2   Woody

3   NameType
4   WoodyCowboy
```

解释

1. 匿名散列中含有一组位于花括号内的键 / 值对。这里将匿名散列赋值给引用 $hashref。
2. 散列指针 $hashref 通过箭头运算符按地址访问散列表。键 Name 与值 Woody 相关联。
3. keys 函数通过引用（指针）返回匿名散列中的所有键。
4. values 函数通过引用（指针）返回匿名散列中的所有值。

13.1.4 嵌套数据结构

创建指向匿名数据结构的引用（指针）将形成更复杂的数据类型。例如，用户可以在散列中嵌套散列，或者创建元素为散列的数组，又或者创建元素为数组的数组等。

和简单引用一样，当按地址访问匿名数据结构时，必须在引用前面加上正确的表示数据类型的特殊符号。例如，如果 $p 是指向标量的指针，则可用 $$p 按地址访问标量内容；如果 $p 是指向数组的指针，则可用 @$p 按地址访问该数组，或者用 $$p[0] 访问数组的第一个元素。用户还可通过块处理的方式按地址访问指针。$$p[0] 也可写作 ${$p}[0] 或 @{p}[0..3]。有时程序需要使用角括号，以避免出现多义性；而且有的时候角括号还是必不可少的，以便能通过特殊字符按地址访问数据结构中的正确部分。

列表的列表。列表中也可含有其他列表或列表集合，该模式常用于创建多维数组时。示例 13.7 和 13.8 把一个引用赋值为含有另一个匿名数组的匿名数组。

示例 13.7

```
    #!/bin/perl
    # Program to demonstrate a reference to a list with a
    # nested list
1   my $arrays = [ '1', '2', '3', [ 'red', 'blue', 'green' ]];
2   for($i=0;$i<3;$i++){
3       print $arrays->[$i],"\n";
    }

4   for($i=0;$i<3;$i++){
```

```
5        print $arrays->[3]->[$i],"\n";
     }
6    print "@{$arrays}\n";
7    print "--@{$arrays->[3]}--", "\n";

(Output)
3    1
     2
     3
5    red
     blue
     green

6    1 2 3 ARRAY(0x8a6f134)
7    --red blue green--
```

解释

1. $arrays 是一个指向四元数组的引用，该数组含有另一个三元数组，其元素包括 red、blue 和 green。
2. for 循环用于获取第一个数组的值，该数组由元素 1、2 和 3 构成。
3. 在这里使用箭头运算符按地址访问 $arrays。
4. 借助第一个 for 循环迭代遍历嵌套的匿名数组。由于该数组是第一个数组的第 4 个元素，且数组下标是从 0 开始的，因此其第一个下标值为 3，第二个下标则引用其中每个元素。
5. 打印嵌套数组的每个元素（red、blue、green）。
6. 通过在含有引用的块之前附加 @ 符号，便可检索并打印匿名数组中的元素。数组的第三个元素是指向另一个匿名散列的引用。打印其地址。
7. 按地址访问第二个嵌套数组，并打印其内容。

示例 13.8

```
(The Script)
     #!/bin/perl
     # Program to demonstrate a pointer to a two-dimensional array.
1    my $matrix = [
                  [ 0, 2, 4 ],
                  [ 4, 1, 32 ],
                  [ 12, 15, 17 ]
                ] ;

2    print "Row 3 column 2 is $matrix->[2]->[1].\n";

3    print "Dereferencing with two loops.\n";
4    for($x=0;$x<3;$x++){
5        for($y=0;$y<3;$y++){
6           print "$matrix->[$x]->[$y] ";
         }
         print "\n\n";
     }
     print "\n";
7    print "Derefencing with one loop.\n";
8    for($i = 0; $i < 3; $i++){
9       print "@{$matrix->[$i]}", "\n\n";
     }
10   $p=\$matrix;     # Reference to a reference
```

```
11  print "Dereferencing a reference to reference.\n"
12  print ${$p}->[1][2], "\n";
(Output)
2   Row 3 column 2 is 15.
3   Dereferencing with two loops.
6   0 2 4
    4 1 32
    12 15 17

7   Dereferencing with one loop.
9   0 2 4
    4 1 32
    12 15 17

11  Dereferencing a reference to reference.
12  32
```

解释

1. 将引用（指针）$matrix 赋值为三个匿名数组中的一个匿名数组，即二维数组（列表的列表）。
2. 通过箭头运算符访问数组中的第一个元素。在相邻的下标方括号之间隐含了一个箭头运算符，该运算符是可选的。它也可写成 $matrix->[2][1]。
4. 进入外层循环。该循环迭代遍历数组的每一行。
5. 进入内层循环。该循环迭代遍历数组的每一列。
6. 通过引用（指针）打印二维数组的每一个元素。
8. 这次只通过一个 for 循环打印矩阵的内容。
9. 使用块格式按地址访问指针。打印每个列表中的所有元素。
10. $p 是一个引用，其值指向另一个引用。它又称作指向指针的指针。
12. 如要访问数组元素（即按地址访问 $p），则应当在前面额外提供一个美元符号，一个用于 p，另一个则用于 matrix。在相邻的下标方括号之间隐含了箭头运算符，亦可写成 $p->[1]->[2]。

散列数组。列表中也可含有一个或一组散列。示例 13.9 将把引用赋值为含有两个匿名散列的匿名数组。

示例 13.9

```
1   my $petref = [   { "name"  => "Rover",
                       "type"  => "dog",
                       "owner" => "Mr. Jones",
                     },
2                    { "name"  => "Sylvester",
                       "type"  => "cat",
                       "owner" => "Mrs. Black",
                     }
3                ];


4   print "The first pet's name is $petref->[0]->{name}.\n";
    print "Printing an array of hashes.\n";
5   for($i=0; $i<2; $i++){
6      while(($key,$value)=each %{$petref->[$i]} ){
7         print "$key -- $value\n";
       }
```

```
        print "\n";
    }
    print "Adding a hash to the array.\n";

8   push @{$petref},{ "owner"=>"Mrs. Crow", "name"=>"Tweety",
                      "type"=>"bird" };

9   while(($key,$value)=each %{$petref->[2]}){
10      print "$key -- $value\n";
    }

(Output)
4   The first pet's name is Rover.
    Printing an array of hashes.
7   owner -- Mr. Jones
    type -- dog
    name -- Rover

    owner -- Mrs. Black
    type -- cat
    name -- Sylvester

    Adding a hash to the array.
10  type -- bird
    owner -- Mrs. Crow
    name -- Tweety
```

解释

1. 将引用（指针）$petref 赋值为含有两个匿名散列的匿名数组地址。
2. 这是列表的第二个元素，它是一个含有键 / 值对的匿名散列。
3. 这是匿名数组的闭合方括号。
4. 使用指针 $petref 按地址访问列表。首先选择数据的第 0 个元素，然后使用箭头运算符选取散列中的键。接着显示与键 name 相关联的散列值。
5. 进入 for 循环，循环遍历数组。
6. 进入 while 循环。每次执行循环时，通过 $petref->[$i] 从引用指向的散列中提取键和值，并分别赋值给 $key 和 $value。
7. 显示键 / 值对。
8. 使用 push 函数将新的散列 @{$petref} 添加到数组中。
9. 进入 while 循环。每次执行循环时，通过 $petref->[$i] 从引用指向的散列中提取键和值，并分别赋值给 $key 和 $value。显示新增的散列。
10. 按地址访问 $petref 后，再按地址访问数组的第二个元素 $petref->[0]，并显示嵌套散列中的每个键 / 值对。

散列的散列。 散列表中亦可含有其他一个或多个散列。在示例 13.10 中，将引用赋值为一个由两个键构成的匿名散列表，其中每个键的关联值本身也是散列（由它们自身的键 / 值对构成）。

示例 13.10

```
    #!/bin/perl
    # Program to demonstrate a hash containing anonymous hashes.
1   my $hashref = {
```

```
2                    Math    => {                      # key
                                 "Anna"  => 100,
                                 "Hao"   => 95,       # values
                                 "Rita"  => 85,
                               },
3                    Science => {                      # key
                                 "Sam"   => 78,
                                 "Lou"   => 100,      # values
                                 "Vijay" => 98,
                               },
4                  };

5   print "Anna got $hashref->{'Math'}->{'Anna'} on the Math test.\n";
6   $hashref->{'Science'}->{'Lou'}=90;
7   print "Lou's grade was changed
    to $hashref->{'Science'}->{'Lou'}.\n";
8   print "The nested hash of Math students and grades is: ";
9   print %{$hashref->{'Math'}}, "\n";   # Prints the nested hash, Math
10  foreach $key (keys %{$hashref}){
11     print "Outer key: $key \n";
12     while(($nkey,$nvalue)=each(%{$hashref->{$key}})){
13        printf "\tInner key: %-5s -- Value: %-8s\n",
                                    $nkey,$nvalue;
       }
    }

(Output)
5   Anna got 100 on the Math test.
7   Lou's grade was changed to 90.
8   The nested hash of Math students and grades is: Rita85Hao95Anna100
11  Outer key: Science
13  Inner key: Lou   -- Value: 90
    Inner key: Sam   -- Value: 78
    Inner key: Vijay -- Value: 98
11  Outer key: Math
13  Inner key: Rita  -- Value: 85
    Inner key: Hao   -- Value: 95
    Inner key: Anna  -- Value: 1005
    Anna got 100 on the Math test.
```

解释

1. 定义匿名数组。它由两个散列键构成，即 Math 和 Science。这两个键关联的散列值本身也是散列（键 / 值对）。将散列的地址赋值给 $hashref。$hashref 是一个硬引用（指针）。
2. Math 是一个键，其关联值是一个嵌套散列。
3. Science 是一个键，其关联值也是一个嵌套散列。
4. 这是匿名数组的闭合花括号。
5. 如需访问 Anna 的成绩，应首先按地址访问键 Math，然后使用箭头运算符和嵌套键 Anna。第二个箭头运算符在这里不是必需的，但它能让代码结构更加清晰。事实上，这里可以不用任何箭头运算符，而是写作 $$hashref{Math}{Anna}。
6. 使用引用 $hashref，用户还可更改或添加散列值。本行将更改 Luo 的成绩。
7. 按地址访问 $hashref，打印新的成绩。

8、9. 将引用 $hashref->Math 放在前面带有 % 的花括号中，通过这种方式打印嵌套散列 Math

的内容。% 表示它是匿名散列，由键和值构成。

10. foreach 循环迭代遍历散列里外层键的列表（由 keys 函数生成）。
11. 打印每个外层键。
12. 由于每个外层键都关联到同属散列的值上，因此可以将 $hashref->{$key} 放在前缀有百分号的块中，从而实现按地址访问引用 $hashref。
13. 打印嵌套的键及其关联的值。

带有值列表的散列的散列。散列可以含有嵌套的散列键，并将这些键关联到一组值列表上去。在示例 13.11 中，给引用赋予两个键，而和这两个键相关联的值也是其他散列的键。嵌套的散列键将依次与匿名的值列表相关联。

示例 13.11

```
(The Script)
    # A hash with nested hash keys and anonymous arrays of values
1   my $hashptr = { "Teacher"=>{"Subjects"=>[ qw(Science Math English)]},
                    "Musician"=>{"Instruments"=>[ qw(piano flute harp)]},
                  };
                  # Teacher and Musician are keys.
                  # The values consist of nested hashes.
2   print $hashptr->{"Teacher"}->{"Subjects"}->[0],"\n";
3   print "@{$hashptr->{'Musician'}->{'Instruments'}}\n";

(Output)
2   Science
3   piano flute harp
```

解释

1. 将指针 $hashref 赋值为一个匿名散列，它由两个键 Teacher 和 Musician 构成。Teacher 键关联的值是另一个匿名散列，后者的键将与另一个含有 Science、Math 和 English 的匿名数组相关联。键 Musician 也是由一个匿名散列构成的，该散列的键是 Instruments，它所关联的值则是一个含有 piano、flute 和 harp 的匿名数组。
2. 如需按地址访问指针，可使用箭头运算符分隔嵌套的键。最后一个箭头运算符指向与 Subjects 相关联的数组 Science 的第一个元素。
3. 如要从与键关联的匿名数组中获得所有的值，则应当在指针及其嵌套的键前面加上 @ 符号，并使用箭头运算符隔开每个键。如果变量没有名称，也可使用带有正确数据类型符号的块来代替其名称。在这里，用花括号包围整个数据结构，从而可将整个块当作一个数组来实现按地址访问。

13.1.5 引用和子例程

匿名子例程。通过在使用 sub 关键字的同时不提供子例程名，即可创建一个匿名子例程。该表达式以一个分号结尾。有关使用匿名子例程的更多知识，请参见第 14.3 节。

示例 13.12

```
(The Script)
    #!/bin/perl
```

```
1    my $subref = sub { print @_ ; };
2    &$subref('a','b','c');
     print "\n";

(Output)
1     abc
```

解释

1. 将标量 $subref 赋值为一个匿名子例程的引用。该子例程的惟一功能是打印保存在 @_ 数组中的参数。
2. 通过引用调用该子例程，并传递给它三个参数。

子例程和按引用传递。 在向子例程传递参数时，程序会把它们发送给子例程，然后保存到 @_ 数组中去。如果有多个参数，如数组、标量和另一个数组，则这些参数都将展开并存入 @_ 数组。用户将很难判断哪里是一个参数的结束位置，哪里又是另一个参数的起始位置。除非用户提供每个数组参数的长度，并把这些长度值压入 @_ 数组中，然后才能根据它们来确定哪里是第一个元素的终止位置等。此外，倘若要传递一个含有 1000 个元素的数组，则上述 @_ 数组将变得极为庞大。因此，传递参数时最简便也最高效的途径莫过于按地址传递，见示例 13.13。

示例 13.13

```
(The Script)
1    @toys = qw(Buzzlightyear  Woody  Bo);
2    $num = @toys;  # Number of elements in @toys is assigned to $num
3    gifts( \$num, \@toys );    # Passing by reference

4    sub gifts {
5        my($n, $t) = @_;    # Localizing the reference with 'my'
6        print "There are $$n gifts: ";
7        print "@$t\n";
8        push(@$t, 'Janey', 'Slinky');
     }
9    print  "The original array was changed to: @toys\n";

(Output)
6,7  There are 3 gifts: Buzzlightyear Woody Bo
9    The original array was changed to: Buzzlightyear Woody Bo Janey Slinky
```

解释

1. 给数组 @toys 赋三个值。
2. 将标量 $num 赋值为数组 @toys 的元素总数（请记住，标量只能保存单个值，因此当把数组赋值给标量时，等效于把数组中的元素总数赋值给了这个标量）。
3. 以两个指针作为参数调用子例程 gifts。
4. 进入子例程。
5. 数组 @_ 中含有两个指针变量。
6. 按地址访问指向标量的指针。该指针指向标量 $n。
7. 按地址访问指向数组的指针。该指针指向数组 @toys。
8. push 函数负责为 $t 指向的数组添加两个新元素。
9. 在退出子例程后，打印数组 @toys 的最新值。

示例 13.14

```
(The Script)
    # This script demonstrates the use of references
    # to pass arrays. Instead of passing the entire
    # array, a reference is passed.
    # The value of the last expression is returned.

1  my @list1=(1 .. 100);
2  my @list2=(5, 10, 15, 20);

3  print "The total is : ",  &addemup( \@list1,  \@list2) ,   ".\n" ;
         #  Two pointers are passed
4  sub addemup {
5     my (  $arr1, $arr2) = @_;
              # @_contains two pointers (references)
6     my ($total);
7     print $arr1,  "\n" ;
8     print $arr2,  "\n" ;

9     foreach $num (  @$arr1, @$arr2){
10        $total+=$num;
      }
13  return $total;  #  The expression is evaluated and returned
   }

(Output)
7  ARRAY (0x8a62d68)
8  ARRAY (0x8a60f2c)
3  The total is:  5100.
```

解释

1. 将数组 @list1 赋值为含有 1 到 100 之间所有数字的列表。
2. 将数组 @list2 赋值为含有数字 5、10、15 和 20 的列表。
3. 调用子例程 addemup，并传递给它两个参数。每个数字前面附加的反斜杠表明传递的是数组的地址（指针）。
4. 声明并定义子例程 addemup。
5. 将指针传递给数组 @_，并分别赋值给两个 my 变量 $arr1 和 $arr2。
6. 声明 my 变量 $total。

7～8. 打印指针的地址。

9. 进入 foreach 循环。数组 @list1 和 @list2 分别按地址访问指针，并逐个创建待处理的数组元素列表。
10. 每当执行循环时，$total1 变量将累计 $total+$num 的和。
11. 将得到的和返回到调用该函数的第 3 行代码。由于对该函数的调用过程是作为 print 函数的一个参数而出现的，因此从子例程返回后，程序将打印出子例程的返回结果。

13.1.6 文件句柄引用

将文件句柄传递给子例程的惟一途径就是通过引用传递。用户可使用 typeglob 为文件句柄创建别名，然后再使用反斜杠运算符创建对这个 typeglob 的引用。

示例 13.15

```
(The Script)
    #!/bin/perl
1   open(README, "/etc/passwd") || die;

2   &readit(\*README);      # Reference to a typeglob

3   sub readit {
4       my ($passwd)=@_;
5       print "\$passwd is a $passwd.\n";
6       while(<$passwd>){
7          print;
        }
    }

9   seek(README,0,0) || die "seek: $!\n";
                      # Reset back to begining of job
(Output)
5   $passwd is a GLOB(0xb0594).
7   root:x:0:1:Super-User:/:/usr/bin/csh
    daemon:x:1:1::/:
    bin:x:2:2::/usr/bin:
    sys:x:3:3::/:
    adm:x:4:4:Admin:/var/adm:
    lp:x:71:8:Line Printer Admin:/usr/spool/lp:
    smtp:x:0:0:Mail Daemon User:/:
    uucp:x:5:5:uucp Admin:/usr/lib/uucp:
    nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
    listen:x:37:4:Network Admin:/usr/net/nls:
    nobody:x:60001:60001:Nobody:/:
    noaccess:x:60002:60002:No Access User:/:
    nobody4:x:65534:65534:SunOS 4.x Nobody:/:
    ellie:x:9496:40:Ellie Quigley:/home/ellie:/usr/bin/csh
9   seek: Bad file number
```

解释

1. 将文件 /etc/passed 附加给文件句柄 README，并以读方式打开它。
2. 调用 readit 子例程。这里通过创建对 typeglob 的引用来传递文件句柄参数。首先，通过星号来 glob 文件句柄符号。然后，通过在 typeglob 前面缀以反斜杠，实现对这个 typeglob 的引用。
3. 定义子例程 readit。
4. @_ 变量中含有引用。将这个引用赋值给本地标量型变量 $passwd。现在 $passwd 就是对文件句柄的引用了。
5. 打印 $passwd 引用值，可见它含有 typeglob（别名）的地址。

6 ～ 7. while 循环中的表达式从文件 /etc/passwd 中读取一行内容，并赋值给变量 $_。然后把这些行打印到屏幕上。循环持续执行，直到读取并打印完毕所有的行。

9. seek 函数重新设定了指向该文件的读指针，并将它放回文件开头位置。

13.1.7 ref 函数

ref 函数用于判断引用是否存在。如果 ref 的参数是一个指针（引用）变量，ref 便会返回该引用指向的数据类型。例如，如果指针指向标量，则返回 SCALAR；若指向一个数组，则返回 ARRAY。

如果其参数不是引用，则返回一个 null 字符串。表 13-1 列出了 ref 函数可能返回的值。

表 13-1 ref 函数的返回值

返回值	含 义
REF	指向指针的指针
SCALAR	指向标量的指针
ARRAY	指向数组的指针
HASH	指向散列的指针
CODE	指向子例程的指针
GLOB	指向 typeglob 的指针

示例 13.16

```
(The Script)
1   sub gifts;     # Forward declaration
2   $num = 5;
3   $junk = "xxx";
4   @toys = qw/Budlightyear Woody Thomas/ ;
5   gifts( \$num, \@toys, $junk );
6   sub gifts {
7       my( $n, $t, $j) = @_;
8       print "\$n is a reference.\n" if ref($n);
        print "\$t is a reference.\n" if ref($t);
9       print "\$j is a not a reference.\n" if ref($j);
10      printf "\$n is a reference to a %s.\n", ref($n);
11      printf "\$t is a reference to an %s.\n", ref($t);
    }

(Output)
8   $n is a reference.
    $t is a reference.
9
10  $n is a reference to a SCALAR.
11  $t is a reference to an ARRAY.
```

解释

1. 向前声明子例程 gifts，以便让 Perl 知道这是一个在程序某处已经定义好的子例程。如果在定义之前已经声明了子例程，则在调用它时无需使用 & 符号。
2. 将标量 $num 赋值为 5。
3. 将标量 $junk 赋值为字符串 xxx。
4. 将一个列表赋值给数组 @toys。
5. 调用子例程 gifts。这里借助前缀反斜杠把前两个变量当作引用传递给子例程。最后一个变量 $junk 不是作为引用传递的。
6. 定义子例程 gifts。
7. 对数组 @_ 进行赋值。在这里，把两个引用（地址）和一个非引用变量分别赋值给 $n、$t 和 $j，并使用 my 函数将它们本地化。
8. 以引用 $n 为参数调用函数 ref。只有当 $n 是一个引用时，本行才可以打印出来。

9. $j 不是引用。ref 函数的返回值为 null。
10. print 函数负责打印 ref 函数返回的数据类型值，即 SCALAR（标量）。
11. print 函数负责打印 ref 函数返回的数据类型值，即 ARRAY（数组）。

13.2 读者应当学到的知识

1. 符号引用和硬引用有什么区别？
2. 什么是 typeglob ？
3. 如何创建一个指向散列的指针？
4. 如何区分一个匿名数组和有名数组？
5. 提供两种按地址访问后面这个指针的途径：$ptr = { 'Name' => 'John'; }
6. 如何按地址访问这个指针（$p = \$x ; ）？
7. 什么是嵌套散列（nested hash）？
8. 如何创建一个二维数组？
9. 按引用传递的优越性有哪些？
10. ref 函数有什么用？

13.3 下章简介

既然读者已经了解了有关指针的知识，下面就可以学习如何在 Perl 中创建对象了。下一章将专门介绍面向对象的 Perl。下一章内容较多，而且如果读者打算使用来自其他库或 CPAN 的模块，那么下一章的内容就显得尤为重要了。读者将学习如何创建对象、设置对象属性、通过方法处理对象，还有如何析构或重用对象。读者还将学到有关 @ISA 与继承、闭合性和垃圾收集等方面的知识。此外，还将学习如何通过 POD（Plain Old Document）为模块提供文档。

练习 13

1. 重新编写 tripper（来自第 11 章），取两个指针作为参数，并在子例程中把这些参数从 @_ 数组复制到两个 my 指针变量里。
2. 创建一个名叫 employee 的散列，含有如下三个键：
   Name
   Ssn
   Salary
   将它们的值赋值为未定义值（undef 是 Perl 提供的一个内建函数）。例如，Name => undef。
   a. 为这个散列创建一个引用。
   b. 使用引用为每个键赋值。
   c. 通过内建的 each 函数与引用，打印散列的键和值。
   d. 打印引用的值；换而言之，打印的是该引用变量包含的内容，而不是它指向的内容。
3. 重新编写上面这个脚本，使得其散列成为一个匿名散列，并将该匿名散列赋值给引用（指针）。通

过引用从散列中删除一个键（使用 delete 函数）。

4. 编写一段含有如下结构的程序：

```
$student = {  Name    => undef,
              SSN     => undef,
              Friends => [],
              Grades  => { Science => [],
                           Math    => [],
                           English => [],
                           }
           };
```

通过指针完成赋值操作，并按如下格式显示输出内容：

```
Name is John Smith.
Social Security Number is 510-23-1232.
Friends are Tom, Bert, Nick.
Grades are:
          Science--100, 83, 77
          Math--90, 89, 85
          English--76, 77, 65
```

第 14 章　面向对象的 Perl

14.1　OOP 范例

14.1.1　回顾包与模块

Perl 5 中新增的一个主要功能就是面向对象的编程（Object-Oriented Programming，OOP）。OOP 的核心是其程序组织的方式。诸如 C ＋＋和 Java 之类的面向对象编程语言都能把数据绑定到变量中，并称之为对象（object）。在英文里，可以把对象表述为一个名词：一个人、一个位置、或一件东西。一只猫、一台电脑以及一个雇员都算是对象。形容词能够描述一个名词。例如，这只猫很胆小，这台电脑很快，这个雇员名叫“John”，等等。在面向对象的语言中，负责描述对象的形容词又称作属性（property），亦可称为特性（attribute）。动词能够描述一个对象能干什么，或者能对这个对象进行怎样的操作。在面向对象语言中，这类动词又称作方法（method）。譬如，猫能吃东西、计算机会崩溃、雇员会工作。在 Perl 中，方法只不过是一种特殊的子例程而已。

对象中的数据一般都是私有（private）的。用户可通过对象中的方法向它发送消息。这些方法往往都是公有（public）的。程序的使用者只有通过这些公有的方法才能访问到对象中的数据。譬如，如果有一个名叫“money”的类，则它的方法可以是 earnit()、findit()、stealit 等等。

将数据和方法封装到一起得到的数据结构称为一个类（class）。在使用面向对象的方法时，Perl 包的功能与类相似。对于 Perl 来讲这已经不是什么新的概念了，因为已经把数据封装到包内了。请回顾一下之前对包的间断讨论，会发现包实际上为程序提供了某种程度上的私有性。每个包都拥有自己独立的符号表，即含有“当前”包中所有符号名字的散列表。这样就可以在包中创建具有自己命名空间的变量和子例程。如果在程序中使用了不同的包，则每个包都会有自己独立的命名空间，从而避免了具有相同名称的变量发生冲突。这种将数据隐藏在包内的思路是 Perl 与生俱来的组成部分，也是面向对象编程的基本原则之一。

每个 Perl 程序至少都含有一个名叫 main 的包，其中的 Perl 语句都是内部编译的。标准 Perl 库则由多个含有包的文件组成。其中大多数的文件都称为模块。模块其实只是可以重用的包。代码的重用是通过继承（inheritance）而实现的；即，包可以从它的父包或基类中继承特征，并对其本身的功能进行扩展或限制。类扩展功能的能力又称为多态（polymorphism）。一般认为数据隐藏（封装）、继承和多态性三者构成了面向对象方法的基本原则。

若要创建一个 Perl 模块，仍应使用关键字 package，其作用域是从声明包的位置直到封装块或

文件的末尾。包一般都对应于一个文件。如果其名称中含有扩展名 .pm，并且模块的第一个字母又是大写的话，Perl 就会把这个文件识别为模块。编译指示（pragma）是另一种特殊的模块（模块名全为小写字母，并且也具有 .pm 扩展名），负责指定编译器的行为方式。不论是称作模块还是称作编译指示的包，都有一些特殊特性不同于普通的包。Perl 5 引入的这些特殊特性，为用户提供了通过抽象的面向对象方式对程序建立模型的能力。读者可以把过程化编程语言当作是面向动作的，而将 OO（面向对象）语言当成是面向对象的。

Tom Christianson 在其 Web 主页“Easy Perl 5 Object Intro”上讨论了 Perl 和对象。他认为人们好像在回避使用一些非常方便的 Perl 模块，因为其中有些内容涉及了对象。Christianson 认为大家不应该害怕这些，因为在 OO 编程时使用其他人提供的模块并不比自己设计实现这些模块来得困难[⊖]。即便读者对在编程时使用 Perl 提供的 OOP 特性不感兴趣，也仍然会碰到那些涉及实用工具对象的 Perl 模块。在通读本章内容后，读者就能对这些模块的工作原理有更为深入的理解。

14.1.2 一些面向对象的专用术语

面向对象的编程是一项内容庞大的科目。有关 OO 编程、设计和方法论的图书数以千计。在 20 世纪 90 年代，为了创建更复杂的软件，许多程序员便从传统的结构化编程语言转向了面向对象的编程语言。本书不是一本有关面向对象的设计或编程的书籍，不过在涉及 Perl 的 OOP 特性之前，我们先了解一下与 OOP 有关的关键词。表 14-1 列出了这些关键词。

表 14-1 OOP 关键词

词语	Perl 含义
数据封装	对用于隐藏数据和子例程，和在包中的效果一样
继承	用于代码的重用，通常来自库，这些库中的包是从其他包继承过来的
多态性	字面上的“多种形式”，特别是提供了扩展类功能的能力
对象	一种引用类型，提供其所属的类；它是一个类的实例
方法	负责操纵对象的特殊子例程
类	含有数据和方法的包
构造函数	负责创建和初始化对象的方法
析构函数	负责销毁对象的方法
set/get 方法	负责向对象存入数据或从对象取出数据的方法

14.2 类、对象和方法

14.2.1 现实世界

假定读者需要建造一幢房子。首先应当在某个特定的地址购买一块地皮。然后会雇请一位建筑

⊖ 如要浏览 Tom Christianson 的 Web 主页，请访问 www.Perl.com/CPAN-local/doc/FMTEYEWTK/easy_objects.html。

师或者购买一套计算机软件，以便设计该房屋。设计时，读者必需确定需要哪种类型、式样、多少个房间、门窗形式，等等。设计完毕后，读者便可雇佣一个工程队开始施工。一旦房子建好，读者就拥有了这幢房子的使用权，可以进入该房屋、进行装修、粉刷墙壁、打扫它、购买家具、清理垃圾、布置风景，总之干什么都行。但是读者必需首先建好这幢房子，然后才能在房里房外干任何想干的事情。此后，读者还可以为房子增加新房间、挂牌出售它甚至拆掉它，等等。鉴于房屋设计图在手，读者亦可找到另外一块地皮，建好另一幢基于相同设计的房子，并可以将它涂成不同的颜色，或者摆放不同的装饰品。甚至还可以建造完全一模一样的两幢房子，二者之间只通过惟一的地址予以区分。

在面向对象的语言中，上面的房子可以视为对象，是一个名词。而房屋样式、房间数、类型等可以当成是描述这个对象的属性，正如一个形容词或动词，譬如粉刷房屋、移入房屋、展示房屋，它们都能描述对象的"行为"。

本节将分析一组示例，并讨论 Perl 是如何创建、管理和销毁对象的。在这之后，读者便会对这些概念有更为清晰的理解。

14.2.2 步骤

本章将详细讨论许多方面的内容。总体而言，为了创建一个名叫对象的新数据类型，并定义它能做什么不能做什么，读者需遵循如下几个步骤：

1. 确定对象内容是什么，能完成什么任务（设计）。
2. 在一种名叫类（class）的包里创建新对象（构造函数）。
3. 描述一个对象；譬如，为它设置属性（形容词）。
4. 获取对象（根据数据类型得到对象）。
5. 定义函数（动词），即对象能干什么，以及能对这个对象干什么（方法）。
6. 使用对象（定义用户接口，又称为方法）。
7. 重用对象（继承）。
8. 销毁对象（把对象清除出内容）。

14.2.3 类和私有性

Perl 中的类（class）就是一种包（package）。这两个术语是可以互换的。类一般位于 .pm 模块中，并且其类名和模块名（除 .pm 扩展名）相同。如果硬要区别二者的话，则类是一种含有方法（method）的包，其中的方法是一类特殊的子例程，负责操纵对象。一个 Perl 类一般由如下几个部分组成（参见图 14-1）：

1. 描述对象的数据。
2. 一个名叫"bless"的函数，负责创建对象。
3. 名叫"方法"的特殊子例程，它知道如何去创建、访问、操纵和销毁对象。

由于类在本质上就是一种包，因此它也具有自己独立的符号表，并且也可以在一个类中通过双冒号（Perl 5）或单撇号（Perl 4）访问另一个类中的数据或子例程。

与其他语言不同，Perl 不会严格监控其模块中的公有 / 私有界限[⊖]。如要保证变量的私有性，请

⊖ Wall,L., and Schwartz, R. L. , *Programming Perl*, 2nd ed., O'Reilly & Associates: Sebastopol, CA, 1998, p.287

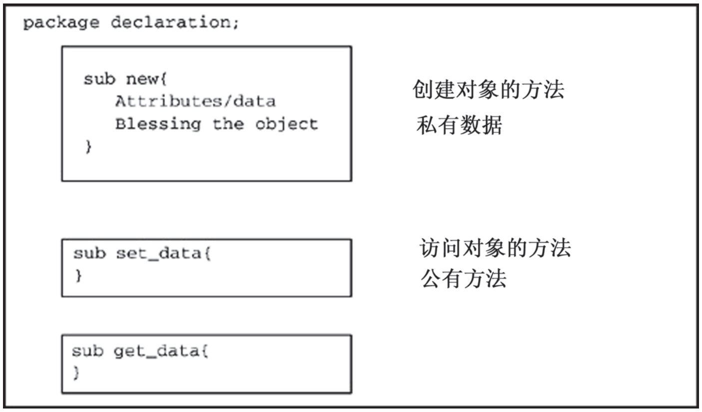


图 14-1 类的组成，.pm 文件

务必使用 my 函数。my 变量只能出现在最内层的封闭块、子例程、eval 或文件。用户不能通过双冒号（或单撇号）从其他包中访问 my 变量，因为 my 变量不和任何包相关，也不存储在创建它的包的符号表里。

示例 14.1

```
    #!/bin/perl
1   package main;

2   $name = "Susan";
3   my $birthyear = 1942;

4   package nosy;
5   print "Hello $main::name.\n";
6   print "You were born in $main::birthyear?\n";

(Output)
5   Hello Susan.
6   You were born in ?
```

解释

1. 包名为 main。
2. 标量 $name 在这个包中的作用域是全局的。
3. 标量 $birthyear 对于该代码块乃至任何封闭的内层代码块而言都是局部变量。
4. 声明 nosy 包。它的作用域是从声明之处知道文件末尾。
5. 通过使用包名与双冒号来限定访问全局变量 $name。
6. 在这个包中，直接通过符号表是不能访问标量 $birthyear 的。因为该标量是通过 my 函数声明在另一个不同的包 main 中的，不和任何符号表相关，而是存储在封闭代码块中创建的专用缓冲区内。

14.2.4 对象

一开始，Perl 中的对象是通过使用硬引用（指针）的方式创建的。引用是负责保存变量地址的

标量，引用就是指针。引用也可指向其他没有名字的变量或子例程，又称匿名变量。例如，这里是一个指向匿名散列的引用 $ref，该匿名散列中含有两个键 / 值对：

```
my $ref = { "Owner"=>"Tom" , "Price" => "25000" };
```

若要访问匿名散列内的值，则可使用箭头运算符按地址访问 $ref 引用，如下所示：

```
$ref -> {"Owner"}
```

为了创建 Perl 对象，首先应当创建引用。一般会把该引用赋值为一个匿名散列的地址（也可将它赋值为其他任意数组或标量的地址）。在这个散列中将含有对象的数据成员和属性。除了存储散列的地址之外，引用还应当获知它所属于的包。为此用户应当先创建引用，然后将其“归入（bless）”到指定的包中。bless 函数能把引用的“东西”（而不是引用本身）加入到包里。它会创建一个内部指针，用于跟踪对象所属的包。对象是归入到类（包）的实例（通常是一个散列表）。如果没有将包作为第二个参数提供给 bless 函数的话，它就会假定使用当前的包。该函数将返回被归入的对象的引用（有关 bless 函数的完整讨论内容请参阅“bless 函数”一节）。

```
my $ref  = { Owner => "Tom",  Price => 250000 };     # This is the object
bless( $ref, Class);    # The object is blessed into the package named Class
return $ref;            # A reference to the object is returned
```

当把对象归入到类中之后，用户就不必使用 @EXPORT_OK 或 @EXPORT 数组去导出对象了。事实上，作为一种通用的规则，如果模块是面向对象的话，则什么也不用导出。

House 类。图 14-2 演示了应当如何虚拟化一个 House 类。首先必须创建一个包。这个包的名字是 House，位于文件 House.pm 中。在 OO 环境中，包亦可称作是类。（请注意，类名必须与不带 .pm 扩展名的文件名相同。）为了表明其封装性，这里用一个 house 对象囊括所有描述数据。属性又称为特性，负责描述对象的特点，譬如对象的属主、样式、尺寸、颜色等等。在这里的示例中，house 的属性包括 Owner、Style 和 Price。在 Perl 中，常常通过一个匿名散列表来描述对象，散列表中的键 / 值对代表着对象的属性。在 house 对象外面，列出了一系列子例程。在面向对象的世界里，这些子例程又称作公有方法（public method），它们为用户提供了访问对象的途径。它们可以用于访问某个对象，但是这么做的前提是这个对象必需是已经存在的。它们往往负责描述对象的“行为”；譬如：该对象能干什么，以及能对该对象做什么。事实上，方法应当是访问对象的惟一途径。对于 house 对象而言，访问它的一个方法可以是移入操作；另一个方法负责清理它；第三个则负责

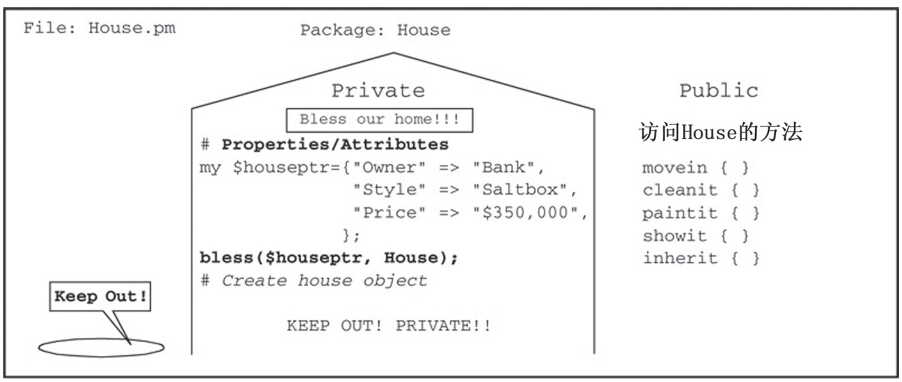


图 14-2　House 类

展示它；等等。在 Perl 中，方法只是一种特别的子例程。

与其他 OO 编程语言不同，Perl 并没有提供特殊关键字 public、private 以及 protected。Perl 的包机制负责创建类，并在类里面保存数据和子例程（即方法）。my 函数使得变量的作用域仅限于本包；而 bless 函数则保证了对象在创建时知道自己属于哪个类。总而言之，对象通常都是一个指向匿名散列表、数组或标量的指针，可通过称作方法的特殊函数予以处理。方法可通过指针来访问指定的对象。

14.2.5 bless 函数

读者可以把 bless 函数看作是负责创建一种名叫“对象”的新数据类型的函数。在图 14-2 中，为了创建 House 对象，首先须借助一个匿名散列赋予它属性值，然后再返回对象的指针，即一个地址。在自己的程序内，读者可把上述地址看作是 house 对象实际所在的内存地址。bless 函数可以直接找到这个地址，并创建一个 House 对象。如果读者归入（bless）一个标量或指针的话，便会把它们转换为一个对象。用专业术语来讲，bless 函数的第一个参数必需是一个指针。借助指向所在包的引用，bless 函数在内部为指针指向的任何内容（即引用目标，referent）都加上了标签。这就是对象的创建流程。如果在第二个函数中没有列出包（类）的话，bless 函数便会把对象标记为属于当前的包。bless 函数使用引用寻找所需的对象，并能返回指向该对象的引用。由于 bless 操作将把对象和特定的包（类）关联起来，因此 Perl 总是能知道各个对象各自属于什么包。用户可以把对象归入到一个类中，然后再重新归入到另一个类，依次类推。但是在同一时刻，一个对象只能属于一个类。

格式

```
bless REFERENCE, CLASSNAME
bless REFERENCE
```

示例 14.2

```
my $reference = {};
return bless( $reference, $class);
```

示例 14.3 说明了如何创建一个对象。首先应创建一个匿名散列，然后把它归入到一个包中，并通过 ref 函数检查该对象是否真的位于这个包中。

示例 14.3

```
(The Script)
1   package House;          # Package declaration

2   my $ref = { "Owner"=>"Tom", #Anonymous hash; data for the package
                "Price"=>"25000", # Properties/attributes
             };
3   bless($ref, House);

    # The bless function creates the object. The hash referenced by
    # $ref  is the object. It is blessed into
    # the package; i.e., an internal pointer is created to keep track
    # of the package where it belongs.
```

```
4   print "The bless function tags the hash with its package
        name.\n";
5   print "The value of \$ref is: $ref.\n";
6   print "The ref function returns the class (package) name:",
        ref($ref), ".\n";

(Output)
4   The bless function tags the hash with its package name.
5   The value of $ref is: House=HASH(0x5a08a8).
6   The ref function returns the class (package) name: House.
```

解释

1. 声明 House 包。这是一个类。
2. 将引用 $ref 赋值为匿名散列的地址，该散列含有两个键 / 值对。这些散列值代表了所述对象的属性值。
3. bless 函数带有一个或两个参数。其中第一个参数是一个引用，第二个参数则是包的名称。如果没有提供第二个参数的话，就假定使用的是当前包[a]。在本示例中，当前包名为 House。现在用户可以看到，这个引用指向的对象位于 House 包中。
5. 打印引用的值（即对象的地址）。它实际上是指向 House 包内一个散列表的引用；简单地讲，它就是指向新的 House 对象的一个指针。
6. 如果 ref 函数的参数是一个指针，并指向某个对象的话，该函数便会返回这个对象所归属到的包的名称。

a. 建议读者尽量使用带有两个参数的 bless 函数，尤其是在需要继承时（详见“继承”一节）。

14.2.6 方法

定义。方法是用于操作对象的子例程。它是一种属于类的特殊子例程，并要求其第一个参数必需是包名或指向对象的引用。这个参数会由 Perl 隐式地赋值。如果没有这个参数，它就和其他的子例程没有分别。方法主要用于创建对象、赋值或更改对象中的数据、或者从对象中检索数据[㊀]。

方法的类型。 方法有两种不同的类型：类（静态）方法和实例（虚）方法[㊁]。类方法取类名为其第一个参数，而实例方法则把对象引用作为第一个参数。

所谓类方法（class method），是一种影响整个类的子例程；例如，它能创建一个对象，或者作用在一系列对象上。类方法需要把类名作为其第一个参数。在面向对象的编程中，构造函数（constructor）就是一种负责创建对象的类方法。在 Perl 中，一般会把这个方法称为 new，当然读者也可任意给它命名。对象的创建过程又称作是对象（或实例）的实例化（instantiation）。

面向对象的程序一般都通过实例方法（也称作访问方法）来控制对象数据的赋值、修改和检索行为方式。只有在对象创建完毕后，才能使用其实例方法。负责创建对象的方法又称作是构造函数。它会返回指向对象的引用。一旦得到了指向新对象的引用（一般命名为 $this 或 $self），实例方法便可通过该引用访问这个对象。实例方法的第一个参数应当是指向对象的引用。然后它会通过该引用来操作指定的对象。

㊀ 与 C ++不同，Perl 并没有为方法的定义提供任何特殊语法。

㊁ 方法类型的称谓在不同书籍上各不相同。Larry Wall 把方法分类为：类方法、实例方法、双性方法（dual-nature method）。

本 PDF 书签由 龙睿·LoRui 制作。
www.LoRui.com

来看看这些东东……

- Yahoo!疯了！
- HP笔记本激活Windows 7

域名转让：

- DaNanPing.com（大南平）
- 15500.net
- Gn7c.com
- DaLongNan.com（大陇南、大龙南）
- 51perl.com（无忧 Perl）
- LoRui.com

调用方法。Perl 提供可调用方法的特殊语法。这里不使用 package::function 形式的语法，而是以另外两种方法来调用方法：类方法调用和实例方法调用。每种方法调用都有两种语法：面向对象语法（object-oriented syntax）和间接语法（indirect syntax）。如果用户要使用对象，则其中任意一种语法都是可以接受的。这里不推荐读者使用老式的带双冒号的方法调用方式。

请读者牢记：和普通的子例程不同，方法总会隐含传递一个参数，该参数可以是类名，也可能是指向对象的引用。譬如，如果用户以三个参数调用某个方法，则实际传递过去的参数会有四个。在面向对象的样式下，其第一个参数值可在箭头的左边看到。

类方法调用

假定方法名为 new，其返回值 $ref 是指向对象的指针。

```
1) $ref = class->new( list of arguments );   # object-oriented syntax
2) $ref = new class ( list of arguments );   # indirect syntax
```

如果类名为 House，则 Perl 会把

```
$ref = House -> new();
```

转换成：

```
$ref = House :: new(House) ;
```

实例方法调用

假定方法名为 display，指向对象的引用名为 $ref。

```
1) $ref->display( list of arguments );       # object-oriented syntax
2) display $ref (list of arguments );        # indirect syntax
```

上述第一个示例使用的方法称作面向对象的方法；而第二个实例用到了箭头运算符，又称作是间接语法。

当 Perl 看到要调用的以上方法时，就知道了这些方法所在的类，因为上述对象都已执行了归入操作（由一个内部指针指明其所在位置）。[⊖]

如果读者调用

```
display $ref ( arguments ...) ;
```

或者

```
$ref -> display(arguments ...) ;
```

其中 $ref 指向 House 类中的一个对象，那么 Perl 就会把上述内容翻译为：

```
House :: display ( $ref , arguments ... ) ;
```

14.2.7 面向对象的模块样式

图 14-3 展示了典型的面向对象模块的布局。创建模块的文件是一个 .pm 文件。在这个示例中，.pm 文件全名为 House.pm。该文件由一个包声明组成。包又可称作是类，因此这就是一个 House

⊖ Perl 调用正确模块函数的能力又称作是运行时绑定（Runtime bunding），来自 Srinivasan, S., *Advanced Perl Programming*, O'Reilly & Associates: Sebastopol, CA, 1997.

类。该类中包含多个子例程，即方法。其中第一种方法的名字叫 new，是构造函数方法。该方法负责定义和创建（构造）指定的对象。当模块使用者调用 new 方法时，就能得到一个指向新建的 House 对象的引用（即 house 对象地址）。new 方法以所在的类名作为其第一个参数。这个方法不仅可以创建对象，还负责归入对象，从而让对象始终知道自己所属的类（包）。（详见“bless 函数”一节。）后两种方法则称作访问方法或实例方法。它们负责保存或读取对象中的数据。在创建好对象实例之前是不能使用这些方法的。（除非房子已经建好，否则你是无法走进或展示它的。）当用户获得了指向对象的引用后，便可使用该引用去调用实例方法。

继续看图 14-3，我们可以看到，对象的数据都是在一个匿名散列中予以描述的（这里亦可使用其他任何数据类型），而对象的地址则赋值给一个私有（my）的引用。（然后通过实例方法 set_data 对对象数据进行赋值。）bless 函数使用对象所属的类名来标记对象，并返回指向对象的指针；即，当模块使用者调用构造函数时，得到的便是指向新对象的引用。构造函数能够“实例化”对象（譬如创建一个 house 对象）。用户可以创建任意多个对象，Perl 会为每个对象提供单独的地址。对象创建完毕后，便可使用实例方法（常称作 setter 和 getter 方法）来操纵这个对象。用户可以调用实例方法去保存或检索对象数据。在调用实例方法时，用户必须提供一个指向该对象的引用，以免访问错误的对象。实例方法通常都把一个指向对象的引用当成其第一个参数。

模块的设计方法有很多种。下图 14-3 只是其中一种简单的途径。

House.pm模块

package *House*;   # *The House class in the House.pm module*

```
sub new{
   my $class = shift;
   my $ref = { };  # define the object
   bless($ref, $class);
   return $ref;
}
```

构造函数方法
创建一个指针
通过bless函数创建一个对象
返回一个指向该对象的引用

```
sub set_data{
   my  $self = shift;
   $self->{key}=value;
}
```

访问方法，或称实例方法
赋值或保存对象的属性

```
sub get_data{
   my $self = shift;
   print $self->{key};
}
1;
```

获得或获取对象的属性

类调用代码

```
#!/usr/bin/perl
# package is main
use House;
$obj1 = House->new(args);# Create the object; a pointer is returned;
           # args refers to any arguments that might be passed
$obj2 = House->new(args);# Create another house object.
$obj1->setdata(args);# Store or assign data to the object
$obj1->getdata(args);# Fetch data from the object

$obj2->setdata(args);# Store or assign data to the object
$obj2->getdata(args);# Fetch data from the object
```

图 14-3　House 类及其调用代码

类构造函数方法。构造函数是一个 OOP（面向对象编程）术语，它指的是一种类方法，负责在类中创建并初始化相应的类对象。构造函数没有特殊的语法。它只是一种让用户获得归属包引用的

方法。Perl 类的第一个方法（即包中第一个子例程）一般都会创建（指向对象的）引用，并将该引用归入到包内。该方法的名字一般都是 new，因为它将建立一个新的“东西”。不过读者亦可将它命名为其他名字，譬如 create、construct、initiate 等等。

一般而言，由 new 子例程生成的对象都是匿名的散列表或匿名数组。用户可以向该匿名散列或数组赋予描述对象性质的数据。这些数据常被称作是对象的属性（property）或特性（attribute）。总体而言，属性定义了对象的状态。

示例 14.4

```
(The Module: House.pm)
1   package House;                  # Class

2   sub new {                       # Class method called a constructor
3       my $class = shift;
4       my $ref={"Owner"=>undef,    # Attributes of the object
                 "Price" =>undef,   # Values will be assigned later
                };
5       bless($ref, $class);
                      # $ref now references an object in this class
6       return $ref;  # A reference to the object is returned

    }
    1;
----------------------------------------
(The User of the Module)
    #!/usr/bin/perl
7   use House;
8   my $houseref = House->new();
    # call the new method and create the object
9   # my $houseref = new House; another way to call the new method
10  print "\$houseref in main belongs to class ",
             ref($houseref),".\n";
```




```
(Output)
10  $houseref in main belongs to class House.
```

解释

1. 声明 House 包。亦可称之为类，因为它含有一个处理对象引用的方法。
2. 子例程 new 在 OOP 术语中又称为是构造函数。构造函数的主要功能是创建并初始化对象。在 Perl 中，构造函数并不包含任何特殊的语法。构造函数是一个类方法，因为它的第一个参数是类的名字。该子例程能把一个经过引用的“东西”（对象）归属到类中，并返回一个指向该对象的引用。该子例程又称作方法，而它处理的“东西”又叫对象。它们所在的包又称作类。
3. 此类子例程接受的第一个参数是包名或类名，在本例中就是 House。这是方法和子例程之间的又一个不同之处。方法的第一个参数必需是一个类名或对象名。
4. 把引用 $ref 赋值为一个匿名散列（对象）的地址。这里把它的键赋值为 undef，意味着目前

它的值尚未定义，将在以后予以定义。

5. 将 $ref 指向的“东西”归入类 $class，并转换为相应的对象。
6. 把一个指向对象的指针返回给构造函数的调用者。
7. 这是另一个使用 House 模块的脚本。在 shbang 行之后，通过 use 语句把模块 House.pm 载入内存。
8. 以包 / 类名 House 作为第一个参数，调用构造函数 new。该函数将返回一个 $houseref 引用，指向一个匿名散列，即新创建的对象。Perl 会隐式地把类名作为第一个参数发送给 new() 方法。
9. Perl 会把 $houseref = $House->new() 翻译为 $houseref = House::new(House)；这一行内容已经注释掉了。它展示了另一种调用方法的途径，即间接语法（indirect syntax）。读者可选用其中任意一种途径。
10. 如果引用已经归入到指定类中，ref 函数就能返回这个类（包）的名字。

类方法和实例方法。回顾：类方法，又称作静态方法，是无需对象实例就能工作的方法或子例程。它们是独立的的函数，代表了类的行为。属于类方法的例子包括负责计算支票数额的函数，以及从数据库中获得姓名列表的函数。最为常见的类方法莫过于构造函数方法。它是负责创建对象的方法。构造函数的第一个参数是类（包）的名字，并能作用在整个类上。

面向对象的程序常常使用访问方法或实例方法来控制对象数据的修改、检索和显示方式。为了处理正确的对象，实例方法（访问方法）需要提供一个对象实例；即对一个已有对象的引用。

如果需要在一个方法中以不同方式表达数据的话，只要提供给用户的接口保持不变，则其他方法就不必受其影响。示例 14.5 中的实例方法用作访问函数，负责显示类里面的数据成员。实例方法取一个对象引用为其第一个参数。在调用实例方法时，请读者注意观察位于 -> 左侧的值。这个值是一个对象，它会隐式地作为第一个参数传递给正在调用的方法。（如果位于 -> 左侧的值是类名的话，则会把类名作为第一个参数传递给方法；譬如，构造函数就是以类名作为其第一个参数的。）

示例 14.5

```
    #!/usr/bin/perl
1   package House;
2   sub new{           # Class/Static method
3      my $class = shift;
4      my $ref={};    # Anonymous and empty hash
5      bless($ref);
6      return $ref;
    }
7   sub set_owner{     # Instance/Virtual method
8      my $self = shift;
9      print "\$self is a class ", ref($self)," reference.\n";
10     $self->{"Owner"} = shift;
    }
11  sub display_owner {
12     my $self = shift;
                  # The object reference is the first argument
13     print $self->{"Owner"},"\n";
    }
    1;

--------------------------------------------------------------------

(The Script)
```

```
    #!/usr/bin/perl
    # The user of the class
14  use House;
15  my $house = House->new;              # Call class method
16  $house->set_owner ("Tom Savage");    # Call instance method
17  $house->display_owner;               # Call instance method

(Output)
9   $self is a class House reference.
13  Tom Savage
```

解释

1. 声明 House 包。亦可称之为 House 类。
2. 类方法 new 是构造函数。它负责创建一个引用，并将其归入 House 类中。
3. 把变量 $class 赋值为类的名字，后者也是构造函数 new 的第一个参数。
4. 把引用 $ref 赋值为一个空的匿名散列。
5. 将 $ref 指向的“东西”（即匿名散列）归入 House 类中。这样一来，该“东西”就变成了一个对象。
6. 从 new 方法把指向对象的引用返回给调用者。
7. 定义实例方法 set_name。
8. 将指向对象的引用从 @_ 数组转移到 $self。现在便可使用实例方法操纵对象了，因为已经获得了对该对象的引用。
9. ref 返回类 House 的名称。只有将对象成功归入后，$ref 才能返回其类名。
10. 从 @_ 数组转移第二个参数。把 Tom Savage 作为一个值赋予 Owner 键。然后使用该实例方法（又叫 setter 方法）将数据赋值给对象。
11. 调用实例方法 display_owner，访问对象内的数据；在本例中，即打印匿名散列中的 Owner 字段值。
12. 该方法的第一个参数是一个指向对象的引用。从 @_ 数组将对象引用转移到 $self 中。
13. 显示键 Owner 的字段值。
14. 这段程序使用了 House.pm 模块。
15. 调用 new 构造函数方法，返回指向一个 House 对象的引用；即指向 House 类中一个匿名散列表的引用。构造函数方法 new 通常至少传递一个参数，也就是类的名字。House->new() 将翻译为 House::new(House)。
16. 以 Tom Savage 为参数调用实例函数 set_owner。请记住，传递给实例方法的第一个参数必需是指向对象的引用，而 Tom Savage 则是第二个参数：

$house->set_owner ("Tom Savage") 会转译为 House::set_owner($house , "Tom Savage");

17. 调用 display 方法。该方法负责显示匿名散列的值，即 Tom Savage。

传递参数到构造函数方法。实例变量常用于在创建对象时对它进行初始化。通过这种方式，便可在每次创建对象时定制其内容。用户可以把描述对象的数据作为参数传递给构造函数方法。之所以把它们称作实例变量（instance variable），是因为程序只有在创建对象或初始化对象时才会生成它们。一般都通过匿名散列或匿名数组来保存实例变量。在下面的示例 14.6 中，该对象“拥有（has a）”或“含有（contains a）”两个实例变量 owner 和 price。

示例 14.6

```
(The Module: House.pm)
1   package House;
2   sub new{           # Constructor  method
3      my $class = shift;
4      my ($owner, $salary) = @_;        # Instance variables
5      my $ref={"Owner"=>$owner,         # Instance variables to
               "Price"=>$price,          # initialize the object
       };
6      bless($ref, $class);
7      return $ref;
    }
8   sub display_object {          # An instance method
9      my $self = shift;        # The name of the object is passed
10     while( ($key, $value)=each %$self){
          print "$key: $value \n";
       }
    }
    1;

------------------------------------------------------------------

(The Script)
    #!/usr/bin/perl
    # User of the class; another program
11  use House;
    # Documentation explaining how to use the House
    # package is called the public interface.
    # It tells the programmer how to use the class.
    # To create a House object requires two arguments,
    # an owner and a price.
    # See "Public User Interface—Documenting Classes" on page 474
    # to create documentation.

    # my $house1 = new House("Tom Savage", 250000);
    # Invoking constructor--two ways.
12  my $house1  = House->new("Tom Savage", 250000);
13  my $house2 = House->new("Devin Quigley", 55000);
    # Two objects have been created.
14  $house1->display_object;
15  $house2->display_object;
16  print "$house1, $house2\n";

(Output)
14  Owner: Tom Savage
    Price: 250000
15  Owner: Devin Quigley
    Price: 55000
16  House=HASH(0x9d450), House=HASH(0xa454c)
```

解释

1. 声明 House 包。
2. 将类方法 new 定义为构造函数。
3. 类方法的第一个参数是类（包）的名字。
4. 从剩余的参数列表中创建实例变量。

5. 将匿名数组的地址赋值给引用 $ref。这里的键都是硬编码的，而值则由实例参数提供。
6. 将引用 $ref 指向的“东西”归入类中，从而得到一个新的对象。
7. 在调用方法时返回引用 $ref 的值。
8. 子程序 display_object 是一个实例方法。在该类中予以定义。
9. 实例方法的第一个参数是指向对象的引用。
10. while 循环负责通过 each 函数从 $self 指向的散列（对象）中获得键和值。
11. 类的用户把 House.pm 加载到命名空间。
12. 以三个参数调用 new 方法：House、Tom Savage 和 250000。其中第一个参数是类的名称。读者看不到这个参数，因为它是由 Perl 隐式发送给构造函数的。这里惟一的要求是把 Owner 值放在第一个参数，而将 Price 值作为第二个参数。这里没有提供任何错误检查机制。该示例只是为了说明如何将参数传递给构造函数。返回到 $house1 的值是一个指向散列对象的引用。
13. 以不同的参数再次调用 new 方法，新的参数是 Devin Quigley 和 550000。返回给引用 $house2 的值是指向另一个对象的引用。这里使用 new 方法创建了两个对象。用户亦可创建任意多个对象。这些对象将各自拥有独立的地址，正如第 16 行的输出内容所示。由于在构造函数中都完成了对象的归入处理，Perl 知道这些对象都位于 House 类中。
14. 调用实例方法，显示 $house1 所引用对象的数据。
15. 再次调用实例方法，显示 $house2 所引用对象的数据。
16. 打印这两个对象的地址。

传递参数到实例方法。 实例方法的第一个参数必需是指向对象的引用。在被调用的方法中，这个值一般都是从 @_ 数组转移而来的，并保存在 my 变量 $self 或 $this 中，事实上变量的名称在这里根本无关紧要。其余参数的处理流程与常规子例程相同。

示例 14.7

```
#!/bin/perl
# Program to demonstrate passing arguments to an instance method.
# When method is called, user can select what he wants returned.
package House;

sub new{     # Constructor, class method
   my $class = shift;
   my ($owner, $salary, $style) = @_;
   my $ref={ "Owner"=>$name,
             "Price"=>$salary,
             "Style"=>$style,
           };
   return bless($ref, $class);
}
sub display {            # Instance method
   my $self = shift;   # Object reference is the first argument
   foreach $key ( @_){
      print "$key: $self->{$key}\n";
   }
}
1;

--------------------------------------------------------------------
```

```
(The Script)
    #!/bin/perl
    # User of the class--Another program
7   use House;
8   my $house = House->new("Tom Savage", 250000, "Cape Cod");
9   $house->display ("Owner", "Style");
                          # Passing arguments to instance method

(Output)
Owner: Tom Savage
Style: Cape Cod
```

解释

1. 声明 House 包。由于这个包含有归入的引用与方法，因此也可称之为一个类。
2. 本行的 new 方法是一个构造函数。该对象是一个匿名散列，含有三个键 / 值对。这些键值对又称为对象的属性。将它们的值作为参数传递给构造函数，并创建一个对象。该对象由 $ref 引用，归入到 House 类中。因此，Perl 能够追踪这些对象属于哪个包。
3. 定义实例方法 display。
4. 实例方法 display 的第一个参数是指向 House 对象的引用。将该参数移出来，并赋值给 $self。
5. foreach 循环负责逐一遍历 @_ 数组中的剩余元素，并依次将每个参数赋值给变量 $key。变量 $key 是一个匿名数组中的关键字。
6. 打印从关联数组中选定的值。
7. 类的调用者将该模块载入到程序中。
8. 调用构造函数方法，创建一个新的 House 对象。
9. 以给定参数调用实例方法。其传递的第一个参数是一个指向对象的指针，即 $house，不过用户是看不到这个参数的。其余参数则由用户自行提供，它们都位于括号内。

命名参数。到现在为止，上面这些实例都使用的 House 对象。在下面的示例中，我们将创建一个新的 Employee 对象。Employee 构造函数将接受一些用于描述雇员属性情况的参数。如果构造函数方法需要按顺序传递姓名、地址和薪水的话，程序将很容易以错误的顺序传递这些参数，从而导致把地址值赋予姓名，或者将姓名值赋予薪水，等等。如果使用了命名变量，就能提供一种方法，可以确保不论以何种顺序把参数传递给方法，都能将参数值赋予正确的属性。用户可通过调用或驱动程序以键 / 值对的形式传递参数，然后由构造函数将这些参数接受为一个散列表。示例 13.8 展示了如何把命名参数传递给方法，以及如何在方法中传递命名参数。

用户 / 驱动程序

示例 14.8

```
    #!/usr/bin/perl
    # User of Employee.pm--See Example 14.9 for module
1   use Employee;
2   use warnings;
    use strict;
3   my($name, $extension, $address, $basepay, $employee);
4   print "Enter the employee's name. ";
    chomp($name=<STDIN>);
    print "Enter the employee's phone extension. ";
```

```
    chomp($extension=<STDIN>);
    print "Enter the employee's address. ";
    chomp($address=<STDIN>);
    print "Enter the employee's basepay. ";
    chomp($basepay=<STDIN>);

    # Passing parameters as a hash
5   $employee = Employee->new( "Name"=>$name,
                               "Address"=>$address,
                               "Extension"=>$extension,
                               "PayCheck"=>$basepay,
                             );
    print "\nThe statistics for $name are: \n";

6   $employee->get_stats;

(Output)
Enter the employee's name. Daniel Savage
Enter the employee's phone extension. 2534
Enter the employee's address. 999 Mission Ave, Somewhere, CA
Enter the employee's basepay. 2200

The statistics for Daniel Savage are:
Address = 999 Mission Ave, Somewhere, CA
PayCheck = 2200
IdNum = Employee Id not provided!
Extension = 2534
Name = Daniel Savage
```

解释

1. 该程序用到了 Employee.pm 模块。
2. 对于可能发生的错误，触发警告信息、strict 编译指示符将跟踪全局和未定义的变量、裸词等。
3. 创建一个词法上私有的变量列表。
4. 要求程序的用户输入想要传递给 Employee 模块的信息。
5. 调用构造函数，并以键 / 值对的形式传递参数；即，在 Employee 中将散列传递给构造函数。（详见示例 14.9。）然后返回转向对象的引用，并将它赋值给 $employee。
6. 调用实例方法 get_stats，显示雇员的属性。

面向对象的模块

示例 14.9

```
    # Module Employee.pm--See Example 14.8 to use this module.
1   package Employee;
2   use Carp;
3   sub new {
4      my $class = shift;
5      my(%params)=@_;    # Receiving the hash that was passed
6      my $objptr={
7          "Name"=>$params{"Name"} || croak("No name assigned"),
           "Extension"=>$params{"Extension"},
8          "Address"=>$params{"Address"},
           "PayCheck"=>$params{"PayCheck"} ||
                   croak("No pay assigned"),
9          ((defined $params{"IdNum"})?("IdNum"=>$params{"IdNum"}):
```

```
                ("IdNum"=>"Employee's id was not provided!"
            )),

        };
10      return bless($objptr,$class);
    }
11  sub get_stats{
12      my $self=shift;
13      while( ($key, $value)=each %$self){
            print $key, " = ", $value, "\n";
        }
        print "\n";
    }
    1;
```

解释

1. 声明 Employee 类（包）。
2. 使用标准 Perl 库中的 Carp 模块处理错误信息。这里没有使用 Perl 内建的 die 函数，而是可以使用来自 Carp 模块的 croak 方法，以便在碰到出错情况时退出程序，并打印有关出错原因的更详细的信息。
3. 定义构造函数方法 new。
4. 构造函数方法的第一个参数是类名。将其移出 @_ 数组，并赋值给 $class。
5. 将 @_ 数组中的其余参数赋值给散列变量 %params。然后以键 / 值对的形式将该变量传递给构造函数。
6. 将引用 $ref 赋值为一个匿名散列的地址。
7. 为键 Name 赋值，并从散列 %params 中检索该值。这里已经完成了出错检查。如果没有为键 Name 提供相应值的话，程序就会执行 croak 函数，以便让用户知道程序并没有给 Name 赋值，然后退出程序。
8. 从散列 %params 中获取参数值，并赋值给 Address 属性。
9. 这个例子说明了如何确保模块的用户传递了正确的参数。条件语句读作：如果散列 %params 中存在定义好的键 IdNum，则获取其值并赋予 IdNum；否则，在程序运行时告诉用户他忘了指定该参数。在使用 croak 函数的示例中，如果用户没有按照要求提供输入的话，程序便会退出执行。而在这种形式的检查中，程序将继续执行。
10. 为属性赋值，然后将指向对象的引用归入到类，并把该引用返回给调用者。
11. 定义实例方法 get_stats。
12. 从 @_ 数组中移出第一个参数，并将它赋值给 $self。该参数是指向对象的指针。
13. 进入 while 循环。each 函数负责返回并打印对象中的每一个键 / 值对。

14.2.8　多态性和动态绑定

Webster 辞典把多态性（polymorphism）定义为：polymorphism: n. 1. the state or condition of being polymorphous。[⊖]

为了说的更清楚些，下面是另一个定义：具有多种形态或采取多种行为方式的能力 ... 同一个操

[⊖] *Webster's Encyclopedic Unabridged Dictionary of the English Language*, Random House Value Publishing: Avenel, NJ, 1996, p.1500. Reprinted by permission.

作在不同的类中表现为不同的行为。

多态性拥有很多种不同的描述方式，它早已隐含在了 OO 语境中。在 Perl 中，它的意思是可以使用同一个名称在不同的类中提供同一种方法，而在调用方法时，它会自动进行正确的选择；换而言之，当通过对象引用调用方法时，它将进入对象所属的类中。

下面这个示例 14.10 说明了多态性的特点。这里用到了两个模块 Cat.pm 和 Dog.pm，每个模块都含有三个同名函数：new、set_attributes 和 get_attributes。示例 14.10 中的驱动程序（或用户程序）将使用这两个模块。当调用各自的类构造函数时，便能返回指向 cat 或 dog 对象的引用。当使用对象引用来访问方法时，Perl 会知道调用的子例程及其所属的类，即使方法出现重名也不要紧。Perl 确定了待调用对象所属的类，并在该类（包）中搜寻调用的方法。这里调用正确方法的能力就展示出所谓的多态性。动态绑定又称为运行时绑定（runtime binding），它能允许程序推迟调用正确的方法，直到程序开始执行。它与多态性一起负责将正确的方法绑定到关联的类上，而不必通过 if 语句来决定需要调用的方法。这样就提供了很大的灵活性，并且这对继承机制的正确性也是非常必要的。

若要利用多态性和运行时绑定，就必须使用面向对象的语法，而不是传统的 :: 语法。例如，倘若由两个类 Director 和 Rifleman，它们均含有方法 shoot。用户可以把它写成 $object->shoot，而 Perl 会知道对象所属的类。它会在编译时确定正确的类。通过这种方式，Director 就不会在分配角色时射出子弹，而 Rifleman 也不会尝试拍摄有关步枪射击的电影。读者还可以添加其他的类，譬如含有不同 shoot 方法的 BasketballPlayer 类，并能确保对于该类也能调用正确的方法。如果没有运行时绑定和多态机制的话，程序就只能根据某种条件判断的结果才能确定该用哪个正确的类，譬如：

```
if ( ref($object1) eq "Director") {
   Director::shoot($object1);
elsif ( ref($object2) eq "Rifleman" ){
   Rifleman::shoot($object2);
else{
   BasketballPlayer::shoot($object3);
}
```

在面向对象的语法中，Perl 会隐式地将指向对象的一个引用作为参数传递给方法。由于 Perl 会把对象引用传递当作第一个参数传递给访问方法，并且该对象也已经归入到合适的类里，因此 Perl 能够实现多态性，并作出正确的选择！假定在示例 14.10 中创建的 $object1 对象属于 Director 类，$object2 对象属于 Rifleman 类，而 $object3 则属于 BasketballPlayer 类。

```
$object1->shoot;evaluates to Director::shoot($object1);
$object2->shoot;evaluates to Rifleman::shoot($object2);
$object3->shoot;evaluates to BasketballPlayer::shoot($object3)
```

示例 14.10

```
( File: Cat.pm)
1   package Cat;
2   sub new{       # Constructor
       my $class=shift;
       my $dptr={};
       bless($dptr, $class);
    }
3   sub set_attributes{             # Access Methods
       my $self= shift;
4      $self->{"Name"}="Sylvester";
```

```
        $self->{"Owner"}="Mrs. Black";
        $self->{"Type"}="Siamese";
        $self->{"Sex"}="Male";
    }
5   sub get_attributes{
        my $self = shift;
        print "-" x 20, "\n";
        print "Stats for the Cat\n";
        print "-" x 20, "\n";
        while(($key,$value)=each( %$self)){
           print "$key is $value. \n";
    }
    print "-" x 20, "\n";
    1;
--------------------------------------------------------------------
(File: Dog.pm)
6   package Dog;
7   sub new{                # Constructor
        my $class=shift;
        my $dptr={};
        bless($dptr, $class);
    }
8   sub set_attributes{
        my $self= shift;
9       my($name, $owner, $breed)=@_;
10      $self->{"Name"}="$name";
        $self->{"Owner"}="$owner";
        $self->{"Breed"}="$breed";
    }
11  sub get_attributes{
        my $self = shift;
        print "x" x 20, "\n";
        print "All about $self->{Name}\n";
        while(($key,$value)= each( %$self)){
           print "$key is $value.\n";
        }
        print "x" x 20, "\n";
    }
    1;
--------------------------------------------------------------------
```

解释

1. 这是模块 Cat.pm 中对一个名叫 Cat 的类的包声明。
2. Cat 类的构造函数方法名为 new。本行会把一个 Cat 对象归入到类中。
3. 访问方法 set_attributes，定义 Cat 对象的数据属性。
4. 使用对象指针 $ref 给 Cat 对象赋予一个键 / 值对，以便指定它的名字。
5. 使用另一个访问方法 get_attributes 显示 Cat 对象。
6. 在另一个文件 Dog.pm 中，对一个名叫 Dog 的类提供包声明。
7. 与 Cat 类一样，Dog 类也有一个名叫 new 的构造函数。本行会把一个 Dog 对象归入到类中。
8. 访问方法 set_attributes，定义 Dog 对象的数据属性。
9. 从驱动程序中传递 Dog 对象的属性，并赋值给 @_ 数组。
10. 使用对象指针 $ref 给 Dog 对象赋予一个键 / 值对，以便指定它的名字。
11. 与 Cat 类一样，使用另一个访问方法 get_attributes 显示 Dog 对象内容。

示例 14.11

```
(The Script: driver program for Example 14.10)
#!/bin/perl
1   use Cat;    # Use the Cat.pm module
2   use Dog;    # Use the Dog.pm module

3   my $dogref = Dog->new;      # Polymorphism
4   my $catref= Cat->new;

5   $dogref->set_attributes("Rover", "Mr. Jones", "Mutt");
6   $catref->set_attributes;    # Polymorphism

7   $dogref->get_attributes;
8   $catref->get_attributes;

(Output)
xxxxxxxxxxxxxxxxxxxx
All about Rover
Owner is Mr. Jones.
Breed is Mutt.
Name is Rover.
xxxxxxxxxxxxxxxxxxxx
--------------------
Stats for the Cat
--------------------
Sex is Male.
Type is Siamese.
Owner is Mrs. Black.
Name is Sylvester.
--------------------
```

解释

1. use 指令负责加载 Cat.pm 模块。
2. use 指令加载 Dog.pm 模块。现在程序便可访问这两个类了。
3. 调用 new 构造函数[a]。其第一个参数是类名 Dog，Perl 会将其翻译为 Dog::new(Dog)。返回的是指向 Dog 对象的引用。Perl 知道这个引用是属于 Dog 类的，因为已经在构造函数中对它完成了归入（bless）处理。创建 Dog 对象的实例。
4. 调用构造函数方法 new，并把类名作为第一个参数传递给它。返回指向 Cat 对象的引用。Perl 会将方法调用翻译为 Cat::new(Cat)。上述两个类都用到了 new 函数，但由于 Perl 知道该方法各自所属的类，因此始终能够调用正确的 new 函数。这就是多态性的一个例子。
5. 有了指向对象的引用后，通过该引用调用访问方法（实例方法）。Perl 会把 $dogref->set_attributes 翻译为 Dog::set_attributes($dogref, "Rover", "Mr. Jones", "Mutt")。
6. 这一次，调用 cat 对象的 set_attributes 方法。Cat 类会设置 cat 的属性。
7. 调用 get_attributes 方法显示 Cat 类的数据属性。
8. 调用 get_attributes 方法显示 Dog 类的数据属性。

a. 这里也可使用间接方法调用 new 构造函数，new Dog。

:: 和 -> 标记。箭头 -> 语法一般用在拥有多态性、动态绑定和继承行为的面向对象程序中。这里也可以使用 :: 语法，但后者更不灵活，而且如果不使用条件语句的话还会出问题。下面的示例中展示了 :: 语法的缺点。如果使用面向对象的语法，就不会出现这些问题了。

示例 14.12

```
    # The Cat class
    package Cat;
    sub new{       # The Cat's constructor
       my $class = shift;
       my $ref = {};
       return bless ($ref, $class);
    }
    sub set_attributes{   # Giving the Cat some attributes,
                          # a name and a voice
       my $self = shift;
       $self->{"Name"} = "Sylvester";
       $self->{"Talk"}= "Meow purrrrrr.... ";
    }
    sub speak {          # Retrieving the Cat's attributes
       my $self = shift;
       print "$self->{Talk} I'm the cat called $self->{Name}.\n";
    }
    1;
--------------------------------------------------------------------
    # The Dog class
    package Dog;   # The Dog's Constructor
    sub new{
       my $class = shift;
       my $ref = {};
       return bless ($ref, $class);
    }
    sub set_attributes{    # Giving the Dog some attributes
       my $self = shift;
       $self->{"Name"} = "Lassie";
       $self->{"Talk"}= "Bow Wow, woof woof.... ";
    }
    sub speak {             # Retrieving the Dog's attributes
       my $self = shift;
       print "$self->{'Talk'} I'm the dog called $self->{'Name'}.\n";
    }
    1;
--------------------------------------------------------------------
    #!/bin/perl
    # User Program
    # This example demonstrates why to use the object-oriented
    # syntax rather than the colon-colon syntax when passing
    # arguments to methods.
    use Cat;
    use Dog;
    $mydog = new Dog;     # Calling the Dog's constructor
    $mycat = new Cat;     # Calling the Cat's constructor

    $mydog->set_attributes;  # Calling the Dog's access methods
    $mycat->set_attributes;  # Calling the Cat's access methods

1   $mydog->speak;
2   $mycat->speak;

3   print "\nNow we make a mistake in passing arguments.\n\n";
```

```
4   Cat::speak($mydog); # Perl goes to the Cat class to find the
                         # method, even though attributes have been
                          # set for the dog!

(Output)
1  Bow Wow, woof woof.... I'm the dog called Lassie.
2  Meow purrrrrr.... I'm the cat called Sylvester.

3  Now we make a mistake in passing arguments.

4  Bow Wow, woof woof.... I'm the cat called Lassie.
```

解释

1. 以面向对象方式调用 speak 方法，保证了 Perl 能将该方法与归属类绑定在一起。

将指向 Dog 对象的引用作为第一个参数传递给 speak 方法。Perl 会把它翻译为 Dog::speak($mydog)。Perl 会在编译时确定使用的类，因此在调用 speak 方法时能把它绑定到正确的类上面去。

2. 将指向 Cat 对象的引用作为第一个参数传递给 speak 方法。Perl 会把它翻译为 Cat::speak($mycat)。
3. 下面的代码会把 dog 引用传递给 Cat 包。如果使用面向对象的方法，则不会出现这种错误，因为 Perl 始终能确定方法所属的类，并调用正确的方法。该特性称为多态性或运行时绑定。
4. 通过使用 :: 标记，Perl 会在运行时识别正确的类，并使用 Cat 类中的 speak 方法，即传送的是指向 Dog 对象的引用。

14.2.9 析构函数和垃圾收集

Perl 维护了对大量对象引用的跟踪信息，并在引用计数达到 0 时自动销毁该对象。在退出程序时，Perl 会销毁与程序相关联的每一个对象，从而处理无用存储单元，并释放其使用过的内容。因此用户无需关心内存清理事宜[⊖]。不过，用户也可在程序中定义 DESTROY 方法，以便在销毁对象前控制该对象。

示例 14.13

```
(The Class)
1 package Employee;
  sub new{
        my $class = shift;
        $ref={};
        bless($ref, $class);
        return $ref;
}

2 sub DESTROY{
        my $self = shift;
3        print "Employee $self->{Name} is being destroyed.\n";
        }

  1;
----------------------------------------------------------------

(The Script)
```

⊖ 如果使用了自引用的数据结构，则读者必须负责破坏该结构。

```
   #!/usr/bin/perl
   # User of the class
4   use Employee;
5   my $emp1 = Employee->new;  # Create the object
6  { my $emp2 = Employee->new;
     $emp2->{"Name"}="Christian Dobbins";
     print "\t\t$emp2->{'Name'}\n";
   } # Create the object

7   my $emp3 = Employee->new;  # Create the object

8   $emp1->{"Name"}="Dan Savage";
    $emp3->{"Name"}="Willie Rogers";

    print "Here are our remaining employees:\n";
    print "\t\t$emp1->{'Name'}\n";
    print "\t\t$emp3->{'Name'}\n";

(Output)
                  Christian Dobbins
 Employee Christian Dobbins is being destroyed.
 Here are our remaining Employees:
                 Dan Savage
                 Willie Rogers
 Employee Dan Savage is being destroyed.
 Employee Willie Rogers is being destroyed.
```

解释

1. 声明 Employee 类，并定义其构造函数方法。
2. 当不再需要某个 Employee 对象时，便会调用其 DESTROY 方法，并打印本行内容。第 6 行出现的对象是在一个代码块中定义的。当退出该代码块时，该对象也应予以销毁。当程序结束时，另一个对象也将销毁。
3. 每当对象得以销毁时，打印本行内容。
4. 这里将使用 Employee 模块。
5. 通过调用构造函数方法，创建一个新的 Employee 对象，得到其对象引用 $emp1。
6. 在同一代码块中创建另一个 Employee 对象。为该对象赋予一个“名字”，即 Christian Dobbins。由于它是一个“my”变量，因此其作用域是受限的。也就是说，在退出该代码块时，这个对象也将予以销毁。这时便会调用 DESTROY 方法，将它清除出内存。
7. 通过调用构造函数方法，创建第三个 Employee 对象，它由 $emp3 来引用。
8. 为 Employee 对象赋予键 / 值对。

14.3 匿名子例程、闭包和私有性

迄今为止，本书所介绍的这些面向对象示例都存在着一个问题：只要用户拿到了指向对象的引用，他就能随心所欲地操纵该对象。即使只希望用户访问模块提供的方法，我们也没有办法阻止他直接访问对象中的数据，因为 Perl 并没有为类数据提供专门的私有区域（private section）。有些批评者或所谓的智者认为，这种缺乏私有性的机制违背了面向对象的理念。不过，针对这个问题，Perl 也提供了很多种解决途径。其中之一就是使用闭包（closure）。

14.3.1 什么是闭包

Larry Wall 把闭包描述成只是一个带有数据（attitude）的匿名子例程[⊖]。Barrie Slaymaker 则把闭包称作“inside-out objects”：对象是附有子例程的数据；而闭包则是附带有一些数据的子例程。

闭包是一类匿名的子例程，即使在定义“my”（词法）变量的代码块之外调用，看上去已经不可能访问得到这些变量的时候，它也能成功访问这些“my”变量。这类子例程会紧紧盯住它所引用的词法变量。每当通过引用调用该子例程时，Perl 会为每次调用分别创建一个全新并且同名的词法变量。该词法变量将持续有效，直到不再有其他引用指向它。

示例 14.14

```
(The Script)
1   my $name="Tommy";

2   {   my $name = "Grandfather";  # Lexical variables
3       my $age = 86;
4       $ref = sub{ return "$name is $age.\n"; }  # anonymous subroutine
    }
5       print "$name is back\n";
6       print &{$ref}; # Call to subroutine outside the block

(Output)
5       Tommy is back.
6       Grandfather is 86.
```

解释

1. 将词法变量 $name 赋值为 Tommy。该变量从本行到文件末尾都是可见的。
2. 进入代码块。将一个新的词法变量 $name 赋值为 Grandfather。该变量的作用域是从本行到代码块末尾。
3. 定义另一个词法变量 $age。该变量的作用域是从本行到包含它的代码块末尾。
4. 定义一个匿名子例程，该子例程与两个词法变量（my 变量）$name 和 $age 位于同一代码块中。把子例程的地址赋值给 $ref。即使在代码块外面调用，该子例程也能访问这两个变量。该子例程可以称作是一个闭包（closure），因为它引用的变量将一直定在里面，直到不再需要为止。
5. 变量 $name 的值 Tommy 现在是可见的。
6. 通过指针 $ref 调用匿名子例程。即使看上去已经不在其作用域内，该词法变量也还是可用的。之所以该变量还在作用域内，是因为该引用还需要访问它们。除非不再引用这些变量，否则 Perl 是不会清除它们的。

示例 14.15

```
(The Script)
    # Closure
1     sub paint {
2        my $color = shift;    # @_ array is shifted
```

⊖ Wall, L., Christianson, T., and Orwant, J., *Programming Perl, 3rd ed.* , O'Reilly & Associates: Sebastopol, CA, 2000, p. 262.

```
3       my $ref = sub {      # Pointer to an anonymous subroutine
4          my $object=shift;
5          print "Paint the $object $color.\n"; # $color still
                                                # in scope
        };
6    return $ref;    # Returns a closure
     }

7    my $p1=paint("red");   #  Creates a closure
     my $p2=paint("blue");  #  Creates a closure

8    $p1->("flower");  # Call to anonymous subroutine
9    $p2->("sky");

(Output)
Paint the flower red.
Paint the sky blue.
```

解释

1. 定义子例程 paint()。
2. 将从 @_ 数组移出的值赋予词法标量 $color。
3. 将标量 $ref 赋值为一个匿名子例程的地址。
4. 匿名子例程从 @_ 数组中获取一个参数。在本例中，当第一次调用子例程时，对象的值是“flower”；第二次调用时，值将变成“sky”。见第 8 行和第 9 行。
5. 本行提供了一个闭包的实例。词法变量 $color 在这里仍位于作用域内。即使子例程 paint() 已经调用并执行完毕，词法变量 $color 也不会离开其作用域，因为匿名子例程将一直持有它。
6. 即便子例程 paint() 已经调用并执行完毕，$color 在这里仍旧是可用的。
7. 子例程 paint() 将返回一个指向匿名子例程的引用。该引用构成了闭包；它会一直持有词法变量（在这里就是 $color），直到不再引用这些变量。
8. 以不同的参数两次调用同一个 paint() 子例程。每次调用 paint() 子例程时，Perl 都会为它创建一个新的词法变量 $color，并赋予该变量独立的值。变量 $color 会包含在返回的那个闭包中。因此 $p1 将含有一个 $color，并初始化为“red”；而 $p2 则含有另一个完全不同的 $color，并将它初始化为“blue”。
9. 指针 $p1 和 $p2 是指向第 3 行所创建的匿名子例程的引用。它们各自形成了一个“闭包”，分别含有在 paint() 中定义的变量 $color，并可以各自访问该变量在本闭包中的副本，直到变量不再被引用为止。

14.3.2 闭包和对象

闭包提供了一种封装对象数据的途径，从而避免用户直接访问到对象的内容。为此，可以定义一个涉及对象数据的构造函数，并提供一个充当闭包的匿名子例程。该匿名子例程将是 set 和 get 对象数据的惟一途径。这里不用归入（bless）对象数据（即将匿名散列归入到类中），而是需要归入匿名的子例程。这时返回的是指向匿名子例程的指针，该指针将成为访问那些在构造函数中定义的私有数据的惟一途径。归入后的匿名子例程将有权访问对象中的私有数据，因为它是在相同的词法作用域内予以定义的。它会把数据和子例程封装到一起，即获得一个闭包。一旦匿名子例程能够引用对象数据的话，就表明这些数据是可供访问的了。

示例 14.6 将演示如何按照如下步骤使用闭包封装对象数据。

1. 为 Student 类定义一个构造函数方法。该构造函数将定义一个负责为每个新 Student 对象设置属性值的空白匿名散列，一个负责跟踪学生总数的全局类变量，以及一个负责封装那些等待赋值与检索的对象数据的匿名子例程。归入操作将返回一个指向匿名子例程的指针。

2. 为该对象定义一些实例方法，用于设置和读取对象数据。这些方法不会直接返回指向对象数据的指针，而是返回指向匿名子例程的指针。它们访问对象数据的惟一途径是以合适的参数调用匿名子例程。

3. 定义析构函数（destructor）方法，负责在销毁每个 Student 对象时显示其内容。

示例 14.16

```
package Student;
sub new { #  Constructor
   my $class = shift;
   my $data={};
   our $students;

   my $ref = sub { # Closure
      my ($access_type, $key, $value) = @_;

      if ($access_type eq "set"){
         $data->{$key} = $value;  # $data still available here
      }
      elsif ($access_type eq "get"){
         return $data->{$key};
      }
      elsif ($access_type eq "keys"){
         return (keys %{$data});
      }
      elsif ($access_type eq "destroy"){
         $students--;
         return $students;
      }
      else{
         die "Access type should be set or get";
      }
      print "New student created, we have ", ++$students,
        " students.\n";
      bless ($ref, $class);  # bless anonymous subroutine
   }
} # End constructor
sub set{
   my ($self,$key,$value) = @_; # $self references anonymous sub
   $self->("set",$key,$value);
}
sub get{
   my ($self,$key) = @_;
   return $self->("get", $key);
}
sub display{
   my $self = shift;
   my @keys = $self->("keys");
   @keys=reverse(@keys);
```

```
        foreach my $key (@keys){
           my $value = $self->("get",$key);
           printf "%-25s%-5s:%-20s\n",$self, $key,$value ;
        }
        print "\n";
     }
17   sub DESTROY{
        my $self = shift;
        print "Object going out of scope:\n";
        print "Students remain: ", $self->("destroy"), "\n";
     }

1;
```

解释

1. 声明 Student 包。
2. 使用构造函数创建其对象。
3. 声明一个指向匿名散列的引用 $data。该引用将用于给对象属性赋值。
4. 使用全局类变量 $student 统计本脚本创建的 Student 对象数目。
5. 这里是一个闭包，即指向匿名子例程的一个引用。该子例程将归入到类中，并负责写入或读取 Student 对象中的数据。
6. 子例程接受三个参数：一个是访问类型，其值可以为“set”、“get”、“key”或“destroy”；一个对象关键字；还有一个对象值。
7. 如果访问类型是“set”的话，即为对象数据设置相应属性值。在调用 set() 方法时，它首先会获得一个指向匿名子例程的指针，然后在第 15 行调用它。
8. 如果访问类型是“destroy”，则调用 DESTROY 方法；即，让一个 Student 对象退出作用域。
9. 每当移除一个对象时，自动把类变量 $student 的值递减去 1。
10. 每当创建一个 Student 对象时，自动把类变量 $student 的值递增 1。
11. 匿名子例程由 $ref 引用。将它归入到 Student 类。
12. 该方法用于把数据放入对象中；即为 Student 对象的属性赋值。
13. 其第一个参数 $self 是一个指向那个归入到类中的匿名子例程的指针；第二个参数是键；第三个则是值。该对象的数据已经“囊括”在匿名子例程中。
14. 匿名子例程在调用时会接受三个参数，分别是：访问类型（即“set”）、一个键、还有一个将赋予对象的值。在为 Student 对象“set”数据时，可供用户使用的惟一途径就是通过上述匿名子例程。它将“一直可用”，直到不再有任何引用指向它为止。
15. get 方法将调用上述闭包，并提供一个访问类型“get”和一个键。程序会把一个值返回给调用者。
16. 该方法用于在 Student 对象中获取数据。它会调用第 5 行提供的闭包，并提供一个键，并据此获得对象的值。这是从对象中“get”数据的惟一途径。
17. 该方法将显示所有的对象数据，为此它首先会调用闭包，获取对象散列中的所有键，然后在一个 foreach 循环中根据键列表获得所有对象数据。
18. 当需要将一个对象移除出内存时（譬如在该对象退出其作用域，或程序执行结束时），Perl 便会调用其特殊的 DESTROY 方法。该方法在调用时将依次调用各个闭包，把访问类型“destroy”发送给这些闭包。详见第 8 行。

模块用户。

示例 14.17

```
  use lib("lib");
1 use Student;

2 $ptr1 = Student->new();  # Create new students
  $ptr2 = Student->new();
  $ptr3 = Student->new();

3 $ptr1->set("Name", "Jody Rogers");  # Set data for object
  $ptr1->set("Major", "Law");

  $ptr2->set("Name", "Christian Dobbins");
  $ptr2->set("Major", "Drama");

  $ptr3->set("Name", "Tina Savage");
  $ptr3->set("Major", "Art");
4 $ptr1->display();  # Get all data for object
  $ptr2->display();
  $ptr3->display();

5 print "\nThe major for ", $ptr1->get("Name"),
      " is ", $ptr1->get("Major"), ".\n\n";

(The Output)
 New student created, we have 1 students.
 New student created, we have 2 students.
 New student created, we have 3 students.

Student=CODE(0x225420)   Name :Jody Rogers
Student=CODE(0x225420)   Major:Law

Student=CODE(0x183192c)  Name :Christian Dobbins
Student=CODE(0x183192c)  Major:Drama

Student=CODE(0x1831a04)  Name :Tina Savage
Student=CODE(0x1831a04)  Major:Art

The major for Jody Rogers is Law.

Object going out of scope:
Students remain: 2
Object going out of scope:
Students remain: 1
Object going out of scope:
Students remain: 0

--------Try to get direct Access----------------------------------
  use lib("lib");
  use Student;

$ptr1 = Student->new();  # Create new students
$ptr2 = Student->new();
$ptr3 = Student->new();
$ptr1->("Name", "Jody Rogers");  # Direct Access Not Allowed
$ptr1->set("Name", "Jody Rogers");
$ptr1->set("Major", "Law");
```

```
-------------------------------------------------
(Output)
New student created, we have 1 students.
New student created, we have 2 students.
New student created, we have 3 students.
Access type should be set or get at Student.pm line 25.
...
```

14.4 继承

继承是指新类可以从已有的类中继承其方法。此外，为了实现类的定制，还可为新类添加代码或修改已有的代码，而不必再重复实现已有的工作。继承的原则是把一个类分为多个共享相同特性的子类，但每一个子类可以提供自己的附加特性，并能将其借用的东西提炼成更为精确的特性。这并不是什么新提出的概念，在学习植物学或动物学时，人们就把生物分为门、界、类、目、族、种和变种。此外，当通过函数把程序的常用要素结合到具体任务中时，也能看到这种思路。

在面向对象的编程中，当编写并调试完成一个类之后，可以将它保存到库中。其他程序员此后便能重用它。程序员还可为现有的类添加特性和功能，而不必重新编写所有的代码。这些都是通过继承机制实现的；即从已有的类派生出新的类。目前流行的 OOP 编程语言都提供了很多可供重复使用的软件模块以及负责组织这些软件的类库。下面我们将介绍 Perl 是如何实现继承机制的。

14.4.1 @ISA 数组和调用方法

@ISA 数组中列出的类（包）表明了当前类的父类（parent class）或基类（base class）。它是 Perl 赖以实现继承的方式。@ISA 数组中含有类（包）的列表，当 Perl 在当前类（包）中无法找到所需方法时，便会在该数组列出的类中查找。如果还是找不到的话，Perl 还会搜索并调用 AUTOLOAD 函数。如果仍然找不到的话，Perl 会在预定义的 UNIVERSAL 包中进行最后的搜索。UNIVERSAL 类是所有包的全局基类，也是类继承机制中位于最顶层的类。

在通常的子例程调用过程中，并不会搜索 @ISA 数组。但如果用户以调用方法的语法来调用子程序的话，程序就会去搜索 @ISA 数组。

示例 14.18

```
#!/bin/perl
# Example of attempting inheritance without updating
# the @ISA array
{ package Grandpa;
  $name = "Gramps";
  sub greetme {
    print "Hi $Child::name I'm your $name from package Grandpa.\n";
  }
}
{ package Parent;
  # This package is empty
}
{ package Child;
  $name = "Baby";
  print "Hi I'm $name in the Child Package here.\n";
```

```
8      Parent->greetme();     # Use method invocation syntax
   }

(Output)
7  Hi I'm Baby in the Child Package here.
8  Can't locate object method "greetme" via package "Parent" at inher2 line 23.
```

解释

1. 声明包 Grandpa。
2. 在包 Grandpa 中将标量 $name 赋值为 Gramps。
3. 定义子例程 greetme。当调用时，它会执行 print 语句。$Child::name 能够引用包中的标量 $name。
4. 声明包 Parent。它是一个空包。
5. 声明包 Child。该包会尝试调用其他包中的方法。尽管这里并没有用到对象和方法，但这个示例主要说明了，如果从这个包不知道的类继承方法的话，会发生怎样的情况。
8. 当 Perl 在包 Parent 中找不到方法 greetme 时，便会打印错误信息。

示例 14.19

```
    #!/bin/perl
    # Example of attempting inheritance by updating the @ISA array
1   { package Grandpa;
       $name = "Gramps";
2      sub greetme {
          print "Hi $Child::name I'm your $name from package Grandpa.\n";
       }
    }

3   { package Parent;
4      @ISA=qw(Grandpa);   # Grandpa is a package in the @ISA array.
                           # This package is empty.
    }

5   { package Child;
       $name = "Baby";
6      print "Hi I'm $name in the Child Package here.\n";
7      Parent->greetme();    # Parent::greetme() will fail
    }

(Output)
6   Hi I'm Baby in the Child Package here.
7   Hi Baby I'm your Gramps from package Grandpa.
```

解释

1. 声明包 Grandpa。
2. 定义子例程 greetme。当调用时，它会执行 print 语句。$Child::name 能够引用包中的标量 $name。
3. 声明包 Parent。
4. 将包 Grandpa 的名字赋值给 @ISA 数组。现在，如果程序试图从 Child 包调用某个 Perl 无法找到的方法，便会尝试在 @ISA 数组列出的 Grandpa 包里搜索它。如果这是一个普通的子例程，无需以调用方法的形式来调用它的话，Perl 就不会去查找 @ISA 数组，因为只有在调用方法时才用得上 @ISA 数组。虽然子例程 greetme 从技术角度来讲根本不算是方法，但只要通过类的方式调用它，Perl 也将会搜索 @ISA 数组。

5. 声明包 Child。
6. 从 Child 包打印该行内容。
7. 在 Parent 包中调用类方法 greetme。@ISA 数组告诉 Perl，如果方法不在 Parent 包内，就在 Grandpa 包中进行搜索。

14.4.2 $AUTOLOAD、sub AUTOLOAD 和 UNIVERSAL

如果无法在当前包或 @ISA 数组中找到子例程（或方法）的话，Perl 就会调用 AUTOLOAD 函数。在 AUTOLOAD 函数中，$AUTOLOAD 变量会赋值为遗漏的那个子例程的名称。所有传递给未定义子例程的参数都将保存在 AUTOLOAD 子例程的 @_ 数组中。如果把 $AUTOLOAD 变量赋值为函数名，那么只要在遗漏子例程的位置提供了 AUTOLOAD 函数，就可以调用该子例程。如果 AUTOLOAD 子例程中用到了 $AUTOLOAD 变量，则用户既可以使用方法形式的调用语法，亦可使用传统的子例程调用语法。如果所有尝试都宣告失败，Perl 还是无法找到所需子例程的话，程序就会在最后的 UNIVERSAL 包（类）中查找遗漏的方法。UNIVERSAL 方法含有所有的类都会继承的三个方法，既 isa()、can() 和 VERSION()。（详见表 14-2。）

表 14-2 UNIVERSAL 方法

方法	用 途	示 例
isa	如果包是从其他包中继承而来的话，返回值为真	Salesman->isa("Employee");
can	如果包及其任意基类中含有给定的方法，则返回真	Salesman->can("get_data");
VERSION	用户检查是否针对该版本加载了正确的模块。在示例中，Perl 调用了 UNIVERSAL 方法 Salesman->VERSION(6.1)	package Salesman; use$VERSION = 6.1 ;

示例 14.20

```
#!/bin/perl
{ package Grandpa;
  $name = "Gramps";
  sub greetme {
   print "Hi $Child::name I'm your $name from package Grandpa.\n";
  }
}

{ package Parent;
  sub AUTOLOAD{
   print "$_[0]: $_[1] and $_[2]\n";
   print "You know us after all!\n";
   print "The unheard of subroutine is called $AUTOLOAD.\n"
  }
}
{ package Child;
  $name = "Baby";
  $AUTOLOAD=Grandpa->greetme();
  print "Hi I'm $name in the Child Package here.\n";
  Parent->unknown("Mom", "Dad");   # Undefined subroutine
}
```

```
(Output)
2   Hi Baby I'm your Gramps from package Grandpa.
10  Hi I'm Baby in the Child Package here.
5   Parent: Mom and Dad
6   You know us after all!
7   The unheard of subroutine is called Parent::unknown.
```

解释

1. 声明包 Grandpa，其中含有子例程。
2. 从 Grandpa 包中打印本行内容。
3. 声明包 Parent，它含有一个 AUTOLOAD 子例程。程序第 11 行会调用未定义的子例程，它含有两个参数 Mom 和 Dad。如果 Perl 在 Child 包中找不到该子例程的话，就会在 @ISA 数组中寻找；如果还找不到，Perl 便会调用 AUTOLOAD 函数进行查找。
4. 定义子例程 AUTOLOAD。
5. 由于这里是以类方法形式调用的该子例程，所以保存在 @_ 数组中的第一个参数是类名。其他两个参数则是 Mom 和 Dad。
6. 打印该行内容，表明程序已经执行到这里。
7. $AUTOLOAD 变量中含有类名和匿名子例程。
8. 声明包 Child。
9. 如果将子例程的名称赋值给标量变量 $AUTOLOAD 的话，Perl 便会自动调用该子例程。
10. 打印该行，表明程序执行的顺序。
11. Child 包尝试访问 Parent 包中的一个方法。在 Parent 包中并没有这个名叫 unknown 的方法或子例程，反而含有一个会在找不到该子例程时自动执行的 AUTOLOAD 子例程。

示例 14.21

```
    #!/bin/perl
1   { package Grandpa;
      $name = "Gramps";
2     sub greetme {
         print "Hi $Child::name I'm your $name from package
             Grandpa.\n";
      }
    }
3   { package Parent;
       # This package is empty
    }
4   { package Child;
    $name = "Baby";
5   print "Hi I'm $name in the Child Package here.\n";
6   Parent->greetme();
    }

7   package UNIVERSAL;
8   sub AUTOLOAD {
9      print "The UNIVERSAL lookup package.\n";
10     Grandpa->greetme();
    }

(Output)
```

```
2   Hi I'm Baby in the Child Package here.
9   The UNIVERSAL lookup package.
5   Hi Baby I'm your Gramps from package Grandpa.
```

解释

1. 声明包 Grandpa。
2. 在这个包中定义子例程 greetme。
3. 声明包 Parent。这是一个空包。
4. 声明包 Child。
5. 打印本行内容，表明程序执行的流程。
6. 把 greetme 子例程当作 Parent 包方法中的一种来调用。
7. 由于在其自身的类或 @ISA 数组中找不到所需方法，并且 Parent 包也没有提供 AUTOLOAD 方法，因此 Perl 会在 UNIVERSAL 包中做最后的努力。这里子例程会调用 greetme 方法。

14.4.3 派生类

正如前面所讨论的，所谓继承（inheritance），就是一个类从已有的类中继承其方法的过程。已有的类又称作是基类（base class）或者父类（parent class），而继承它的类则称作是派生类（derived class）或子类（child class）。基类的所有功能均可由派生类继承，而派生类的功能往往会超出这些功能。如果派生类继承自单个基类，则称为单一继承（single inheritance）。例如，孩子继承父亲的绘画天赋算是现实生活中的一种单一继承。如果某个派生类继承自多个基类，则称为多重继承（multiple inheritance）。譬如小孩从父亲处继承了绘画的天赋，同时又从母亲那里继承了唱歌的天赋。在 Perl 中，派生类可以继承基类（包）中的方法，并可在必要时添加和修改这些方法（参见图 14-4）。

若要继承一个类，通常可以把类名放在 @ISA 数组中。此外，用户可以在 @EXPORTER 或 @EXPORTER_OK 数组中规定可以导出到其他模块的方法。其数据本身则通过内部赋值的匿名散列中的键和值而实现继承。这些变量又称作实例变量，均在构造函数方法中予以定义。

在第 12 章中，已经介绍了标准 Perl 库提供的模块和用户自己创建的模块。为了将模块或编译指示符放到程序中，用户可以以模块名（去除 .pm 扩展名）为参数调用 use 函数。模块具有符号导出能力，能够把自身符号导出到其他需要使用该模块的包。特殊模块 Exporter.pm 负责处理在模块间导入和导出符号的实现细节。用户也应当将该模块置入 @ISA 数组中。如果模块起到类的作用，则在调用其方法时无需显式地把它们列举到 @EXPORT 数组里。请注意，在下面的示例中，类方法和实例方法都没有导出。

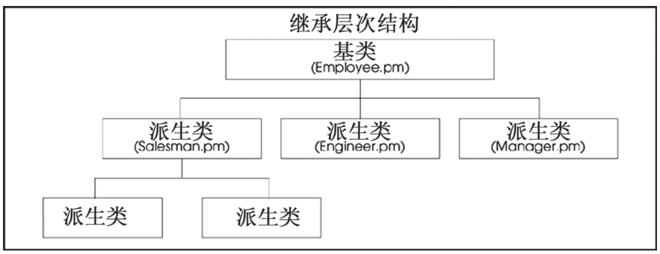


图 14-4　从基类派生出子类

下面这个示例演示了继承。示例 14.22 是 Salesman.pm 的使用者。调用程序无需对基类

Employee.pm 做任何引用（详见示例 14.23）。Salesman.pm 类是从 Employee.pm 类派生得到的。Salesman.pm 类会“使用”Employee.pm 类。

示例 14.22

```
# The Driver (user) Program
1   use Salesman;
2   use strict;
    use warnings;
    # Create two salesman objects
    print "Entering data for the first salesman.\n";
3   my $salesguy1=Salesman->new;
4   $salesguy1->set_data;
5   print "\nEntering data for the second salesman.\n";

6   my $salesguy2=Salesman->new;
7   $salesguy2->set_data;

8   print "\nHere are the statistics for the first salesman.\n";
9   $salesguy1->get_data;
10  print "\nHere are the statistics for the second salesman.\n";
11  $salesguy2->get_data;

(Output)
    The salesman is an employee.
    The salesman can display its properties.
    Entering data for the first salesman.
    Enter the name of the employee. Russ Savage
    Enter the address of Russ Savage. 12 Main St., Boston, MA
    Enter the monthly base pay for Russ Savage. 2200
    Before blessing package is: Employee
    After blessing package is: Salesman
    Enter Russ Savage's commission for this month. 1200
    Enter Russ Savage's bonuses for this month. 500.50
    Enter Russ Savage's sales region. northeast

5   Entering data for the second salesman.
    Enter the name of the employee. Jody Rodgers
    Enter the address of Jody Rodgers. 2200 Broadway Ave, Chico, CA
    Enter the monthly base pay for Jody Rodgers. 34500
    Before blessing package is: Employee
    After blessing package is: Salesman
    Enter Jody Rodgers's commission for this month. 2300
    Enter Jody Rodgers's bonuses for this month. 1400
    Enter Jody Rodgers's sales region. northwest

8   Here are the statistics for the first salesman.
    Name = Russ Savage.
    Bonuses = 500.50.
    Commission = 1200.
    BasePay = 2200.
    Address = 12 Main St., Boston, MA.
    PayCheck = 3900.5.
    Region = northeast.
```

```
10  Here are the statistics for the second salesman.
    Name = Jody Rodgers.
    Bonuses = 1400.
    Commission = 2300.
    BasePay = 34500.
    Address = 2200 Broadway Ave, Chico, CA.
    PayCheck = 38200.
    Region = northwest.
```

解释

1. use 指令负责把模块 Salesman 加载到 main 包中。含有用户界面的程序通常又称为驱动程序（driver program）。它使用 Salesman 模块创建新的 Salesman 对象。尽管 Salesman 类是从 Employee 类派生出来的，但这个程序对此却一无所知。该程序需要做的事情就是使用正确的参数（如果有的话）调用 Salesman 构造函数，并要求用户提供输入。
2. 打开 strict 和 warning 编译指示，从而避免程序使用不安全的结构，并产生编译和运行时报错信息。
3. 调用 Salesman 类的构造函数，返回指向 salesman 对象的一个引用 $salesguy1。
4. 调用 set_data 方法，要求用户输入第一个销售员的信息。从 Salesman.pm 模块中调用该方法。
5. 调用 Salesman 模块的构造函数方法。返回一个指向新的 Salesman 对象的引用，并将它赋值给 $salesguy2。
6. 针对第二个 salesman 对象调用 set_data 方法。同样从 Salesman.pm 模块中调用该方法。
7. 在这里使用继承。调用 get_data 方法，但并不在 Salesman 模块中实现该方法。Perl 会在 Salesman 模块中搜索 @ISA 数组，并获得其列出的 Employee 基类。由于后者提供了 get_data 方法，因此程序会调用该方法。显示第一个销售人员的统计数字。
8. 针对第二个销售员，调用 get_data 方法。

示例 14.23

```
    # Module Employee.pm
    # The Base Class
1   package Employee;
2   use strict;
3   use warnings;
    # Constructor method
4   sub new {
5      my $class = shift;
6      my $self = {_Name=>undef,
                   _Address=>undef,
                   _BasePay=>undef,
                  };
7      return bless($self, $class);
    }

    # Instance/access methods
8   sub set_data{
9      my $self=shift;
10        print "Enter the name of the employee. ";
11        chomp($self->{_Name}=<STDIN>);
          print "Enter the address of $self->{_Name}. ";
          chomp($self->{_Address}=<STDIN>);
          print "Enter the monthly base pay for $self->{_Name}. ";
```

```
            chomp($self->{_BasePay}=<STDIN>);
         }
12 sub get_data{
13    my $self=shift;
14    my ($key,$value);
15    print "Name = $self->{_Name}.\n";
16    while(($key,$value)=each(%$self)){
17       $key =~ s/_//;
18       print "$key = $value.\n" unless $key eq "Name";
      }
      print "\n";
   }
   1;
```

解释

1. 使用与文件 Employee.pm 相同的名称（去除其扩展名）声明包，也就是基类。所有的雇员都有一些共同的特征。在本例中，他们都拥有姓名、地址和薪金。
2. strict 编译指示表明将执行严格的错误检查。换而言之，就是要避免出现不安全的结构，譬如全局变量、裸词等。
3. warning 编译指示针对所有可能发生的错误发出警告信息。
4. 定义 Employee 构造函数方法。这里依照惯例把它命名为 new。
5. 把类名从 @_ 数组中移出来。这是一个类方法，因为它的行为是针对类的，并且不要求提供对象的实例。
6. 通过一个匿名散列以键 / 值对的形式为对象的特性（attribute）/ 属性（property）赋值。如果其值尚未定义，则可以稍候再进行赋值。返回指向散列的引用，并将它赋值给 $self（出现在键前面的下划线是一种命名惯例，用于表明这是对象的私有数据）。
7. 将 $self 引用的对象归入到 Employee 类，并把指向它的引用返回给调用者。
8. 子例程 set_data 是 Employee 类的访问方法。它负责在调用时给对象数据赋值；即向匿名散列添加相应值。
9. 从 @_ 数组移出指向对象的引用，并赋值给 $self。
10. 要求模块调用者提供输入。
11. 通过引用 $ref 为键 _Name、_Address、_BasePay 赋值。
12. 使用访问方法检索并显示对象数据。
13. 从 @_ 数组移出指向对象的引用，并赋值给 $self。
14. 声明两个词法变量。
15. 由于散列值是以随机顺序存入的，因此需要通过本行来保证首先显示雇员的姓名。
16. 进入 while 循环，each 函数通过指向对象的引用（$%self）从对象中提取数据。
17. 从键中去除前缀下划线。
18. 显示对象的其余属性。

示例 14.24

```
   # The Derived Class
1  package Salesman;
2  use strict;
   use warnings;
3  BEGIN{unshift(@INC, "./Baseclass");};
```

```
4    our @ISA=qw( Employee);
5    use Employee;
6    print "The salesman is an employee.\n"
             if Salesman->isa('Employee');
7    print "The salesman can display its properties.\n"
             if Salesman->can('get_data');
8    sub new {          # Constructor for Salesman
9        my ($class)= shift;
10       my $emp = new Employee;
11       $emp->set_data;
12       print "Before blessing package is: ", ref($emp), "\n";
         bless($emp, $class);
13       print "After blessing package is: ", ref($emp), "\n";
         return $emp;
     }
14   sub set_data{
         my $self=shift;
15       my $calc_ptr = sub{ my($base, $comm, $bonus)=@_;
                         return $base+$comm+$bonus; };
16       print "Enter $self->{_Name}'s commission for this month. ";
         chomp($self->{_Commission}=<STDIN>);
         print "Enter $self->{_Name}'s bonuses for this month. ";
         chomp($self->{_Bonuses}=<STDIN>);
         print "Enter $self->{_Name}'s sales region. ";
         chomp($self->{_Region}=<STDIN>);
17       $self->{_PayCheck}=&$calc_ptr( $self->{_BasePay},
                                          $self->{_Commission},
                                          $self->{_Bonuses}
                                        );
     }
     1;
```

解释

1. 声明包（类）Salesman。
2. 载入 strict 和 warning 编译指示。
3. 使用 BEGIN 代码块确保在编译时及时更新 @INC 数组，以便能从中查找基类。Employee.pm 模块位于子目录 Baseclass 中。
4. @ISA 目录中含有该包所用的包名。Employee 模块是 Salesman 模块需要的基类。our 函数使得 @ISA 数组成为一个词法上的全局数组。如果不使用 our 的话，strict 编译指示将导致编译器退出执行该脚本，因为它不允许出现全局变量（除非使用包名和双冒号完全限定其名称）。
5. use 函数负责把 Employee 模块加载到程序中。
6. 所有的包 / 类都是继承自超类 UNIVERSAL 的，该类提供了 isa 方法。如果 Salesman 模块继承了 Employee 模块的话，isa 方法的返回值就是真。
7. 所有的包 / 类都继承了超类 UNIVERSAL，该类提供了 can 方法。如果 Salesman 模块或者其父类含有方法 get_data 的话，则 can 方法就能返回真，否则就返回 undef。
8. 为 Salesman 类定义构造函数方法 new。
9. 传递的参数取自 @_ 数组。将类名赋值给 $class。
10. 调用 Employee 基类的构造函数 new。返回指向对象的引用 $emp。
11. Salesman 类的构造函数调用 set_data 方法来为对象添加属性值。
12. 在将对象归入到类之前，ref 函数会返回该对象所属类（包）的基类的名称，即 Employee。

现在，将拥有新属性值的对象归入到 Salesman 类中。
13. 在将对象归入到类之后，ref 函数会返回该对象所归入的类名，即 Salesman。然后将归入的对象引用返回给调用者。
14. 为 Salesman 类定义实例方法 set_data。
15. 创建用于计算薪金的匿名子例程；返回一个引用。子例程不应由类的使用者调用。只有这个类才能计算薪金数额。
16. 为 Employee 类添加新的属性。
17. 调用计算薪金的子例程。

14.4.4 多重继承

如果一个类继承自多个基类或父类，则称之为多重继承（multiple inheritance）。在 Perl 中，可以通过给 @ISA 数组添加多个类来实现多重继承。

```
package  Child;
@ISA = qw (Mother Father Teacher);
```

上述搜索过程是深度优先的。也就是说，Perl 首先会搜索 Mother 及其继承的类层次，然后是 father 及其继承的类层次，最后再搜索 Teacher 及其所有祖先类。

14.4.5 重写父类方法

有时两个类可能拥有名字相同的方法。如果派生类含有与一个基类方法名称相同的方法，则该方法的优先级将高于基类中的同名方法。如果要跳过派生类中的方法而直接访问基类方法，则必须使用类名和双冒号来确定基类的方法名。

示例 14.25

```
1   package Employee;  # Base class
    use strict;
    use warnings;
    sub new {          # Employee's constructor is defined
       my $class = shift;
       my %params = @_;
       my $self = { Name=>$params{"Name"},
                    Salary=>$params{"Salary"},
                  };
       bless ($self, $class);
       }
2   sub display {       # Instance method
       my $self = shift;
       foreach my $key ( @_){
3         print "$key: $self->{$key}\n";
       }
4      print "The class using this display method is ",
            ref($self),"\n";
    }
    1;
--------------------------------------------------------------------
5   package Salesman;  # Derived class
```

```
    use strict;
    use warnings;
    use Employee;
6   our @ISA=qw (Exporter Employee);
7   sub new {          # Constructor in derived Salesman class
       my $class = shift;
       my (%params) = @_;
       my $self = new Employee(%params);  # Call constructor
                                          # in base class
       $self->{Commission} = $params{Commission};
       bless ( $self, $class );     # Rebless the object into
                                    # the derived class
    }
    sub set_Salary {
       my $self = shift;
       $self->{Salary}=$self->{Salary} + $self->{Commission};
    }
8   sub display{        # Override method in Employee class
       my $self = shift;
       my @args = @_;
9      print "Stats for the Salesman\n";
       print "-" x 25, "\n";
10     $self->Employee::display(@args);  # Access to the
                                         # overridden method
    }
    1;
    -----------------------------------------
    # User or Driver Program
    #!/bin/perl
11  use Salesman;
    use strict;
    use warnings;
12  my $emp = new Salesman ( "Name", "Tom Savage",
                             "Salary", 50000,   # Call to constructor
                             "Commission", 1500,
                           );
    $emp->set_Salary;  # Call to the access method
13  $emp->display( "Name" , "Salary", "Commission");
                       # Call Salesman's display method

(Output)
9   Stats for the Salesman
    -------------------------
    Name: Tom Savage
    Salary: 51500
    The class using this display method is Salesman
```

解释

1. 声明类 Employee，它含有构造函数方法 new 和实例方法 display。
2. 定义 Employee 类中的 display 访问方法。
3. 显示雇员的属性。
4. ref 函数将返回该对象归入到的类名。
5. 声明 Salesman 类。它继承了 Employee 类，又称作派生类。
6. @ISA 数组中含有所需的类名：Exporter 类和 Employee 基类。

7. 这是 Salesman 的构造函数。
8. 派生类的 display 方法优先级更高，因为它属于 Salesman 类。在负责为 Salesman 对象传递参数的驱动程序中调用它。
9. 打印出来的内容来自于派生类（Salesman）的 display 子例程。
10. 通过把方法名限定为类 Employee 中的方法，这里的 display 方法将重写 Salesman 包中现有的 display 方法。
11. 这是驱动程序，使用了 Salesman 模块。
12. 创建新的 Salesman 对象，并使用间接方法调用其构造函数。
13. 调用 display 方法。由于在 Salesman 类中已经提供了 display 子例程，因此它就是要调用的方法。

14.5　公共用户接口：文档类

为了创建一个有用的类，关键的一步就是要为用户提供描述如何使用这个类的文档，又称作公共用户接口（Public User Interface）。不论模块是否是面向对象的，都必须提供某种程度的公共用户接口（即编写好的文档），以便描述程序（客户）应当如何使用这个类（例如，应当为方法传递什么样的参数）。即使类里面的内容发生了改变，公开定义的这些接口也是不应该变的。Perl5 引用了 pod 命令，用于编写模块文档。它通过在子例程中散置 POD（Plain Ole Document）指令来实现其功能。该指令类似于文本或文件中嵌入的 HTML 或 nroff 指令。接着，程序在运行时会通过一个 Perl 过滤程序，以多种不同的格式将命令转译为使用手册页面内容。

14.5.1　pod 文件

如果仔细查看标准 Perl 库的话，就会发现它的模块中含有一些负责解释模块用途和如何使用模块的文档。这些文档可以嵌入在程序内部，也可能位于程序模块的 _END_ 特殊实量之后。该文档又称为 pod，是纯 Ole 文档（Plain Ole Document）的缩写。pod 文件只是一种嵌入了特殊命令的 ASCII 文本文件，这些命令将由 Perl 的特殊解释程序（包括 pod2html、pod2latex、pod2text、pod2man 等）予以解释，其目的是为了创建可通过多种形式表达的格式化文档。UNIX 中的 man 页面就是一种使用 nroff 指令格式化而来的文档实例。用户现在可以方便地将一组 pod 格式指令嵌入到脚本中，并通过下列四种方式之一提供文档：文本、HTML、LaTex 或者 nroff。

pod 文档的第一行必须以等号（＝）开头。以等号开头的每一个单词都是 pod 翻译器的格式化指令。在每个格式化指令后必须紧跟一个空行。

示例 14.26

```
(Here is the documentation found at the end of the BigFloat.pm module in the
standard Perl library, under the subdirectory Math.)

=head1 NAME

Math::BigFloat - Arbitrary length float math package

=head1 SYNOPSIS
```

```
  use Math::BigFloat;
  $f = Math::BigFloat->new($string);
  $f->fadd(NSTR) return NSTR              addition
  $f->fsub(NSTR) return NSTR              subtraction
  $f->fmul(NSTR) return NSTR              multiplication
  $f->fdiv(NSTR[,SCALE]) returns NSTR     division to SCALE places
  $f->fneg() return NSTR                  negation
  $f->fabs() return NSTR                  absolute value
  $f->fcmp(NSTR) return CODE              compare undef,<0,=0,>0
  $f->fround(SCALE) return NSTR           round to SCALE digits
  $f->ffround(SCALE) return NSTR          round at SCALEth place
  $f->fnorm() return (NSTR)               normalize
  $f->fsqrt([SCALE]) return NSTR          sqrt to SCALE places

=head1 DESCRIPTION

All basic math operations are overloaded if you declare your big floats as

   $float = new Math::BigFloat "2.123123123123123123123123123123123";

=over 2

=item number format

canonical strings have the form /[+-]\d+E[+-]\d+/ . Input values can have inbedded
whitespace.

=item Error returns 'NaN'

An input parameter was "Not a Number" or divide by zero or sqrt of negative number.

=item Division is computed to

C<max($div_scale,length(dividend)+length(divisor))> digits by default.
Also used for default sqrt scale.

=back

=head1 BUGS

The current version of this module is a preliminary version of the real thing that
is currently (as of perl5.002) under development.

=head1 AUTHOR

Mark Biggar

=cut
```

解释

上述文本是一个 pod 文件，它含有以等号开头的行以及各种 pod 命令，然后是空白行、文本内容。Perl 提供了特殊翻译器程序，能够读取 pod 文件，将其转译为可读的文件，并格式化为纯文本、HTML、nroff 文办或 Latex 文本格式。下一节将详细描述如何使用 pod 命令进行转译。

14.5.2 pod 命令

把 pod 命令嵌入文本文件是一项非常容易的工作。用户只需将命令放在行首，以 =pod（或其他

pod 命令）开头，并以 =cut 结尾即可。编译器会忽略第一个 =pod 指令直到相应 =cut 指令之间的所有内容，就好像忽略注释一样。这里使用命令的好处是它们允许创建粗体、斜体格式，或者创建含有纯文本的标题（heading）格式。表 14-3 列出了所有的指令。

表 14-3 pod 命令

段落命令	用　途
=pod	标明 pod 的开始，等号及其后面的 pod 命令表明文档的起始位置
=cut	标明 pod 的结束位置
=head1 heading	创建一级标题
=head2 heading	创建二级标题
=item *	开始一个项目列表
=over N	跳过 N 个空格，通常设定为 4 个
=back	将缩进返回为默认值，即没有缩进
格式命令	**用　途**
I<text>	斜体文本
B<text>	粗体文本
S<text>	含有非中断空格的文本
C<code>	含有键入的文本内容，即实际源代码内容
L<name>	创建指向名字的链接（交叉引用）
F<file>	用于列出文件名
X<index>	下标
Z<>	零宽度字符
过滤器专用命令	**用　途**
=for	针对 HTML 的特定命令；譬如：=for html <IMG SRC=file.gif> <B>Figure a.>/B>> 针对文本的特定命令；譬如：=for text 此处文本表示上面图像的含义 针对 manpage 的特殊命令，譬如：=for man .ce3 <center next three lines>

14.5.3 如何使用 pod 解释器

Perl 发布包中已经自带了 pod 解释器。它位于 Perl 主目录的 bin 子目录下：譬如在 /usr/bin/perl5/bin 目录下。

四个解释器分别是：

pod2html （转译为 html）

pod2text （转译为纯文本）

pod2man （转译为 nroff，类似于 UNIX 的 man 页）

pod2latex （转译为 Latex）

若要使用 pod 解释器，最简单的方法就是把要用的解释器拷贝到自己的目录中，譬如：

```
$ cp /usr/bin/Perl5/bin/pod2text
```

此外还应把库例程拷贝到自己的目录中：

```
$ cp /usr/bin/Perl5/lib/BigFloat.pm
```

现在列出目录内容时，就应当出现 pod 解释器和库例程文件。

```
$ ls
BigFloat.pm
pod2text
```

14.5.4 将 pod 文档转译为文本

若想把 pod 文档转译为文本并显示到终端屏幕上，最简便的方法就是使用 Perl 发布包自带的 perldoc 命令。该命令可能不在搜索路径中，不过它一般都位于 Perl 的 bin 目录内。下面这个命令将显示 BigFloat.pm 模块的所有文档。

```
perldoc Math:: BigFloat
```

另一种把 pod 指令转译为文本的方式则是让 pod 解释器筛选整个模块，并创建一个存有转译结果的文本文件。如果用户不把它重定向到输出文件的话，它就只会输出到屏幕上。

```
$ pod2text BigFloat.pm > BigFloat.Text
$ cat BigFloat.Text  (The output file after pod commands have been
                      translated into text.)

NAME
    Math::BigFloat - Arbitrary length float math package

SYNOPSIS
      use Math::BogFloat;
      $f = Math::BigFloat->new($string);

      $f->fadd(NSTR) return NSTR              addition
      $f->fsub(NSTR) return NSTR              subtraction
      $f->fmul(NSTR) return NSTR              multiplication
      $f->fdiv(NSTR[,SCALE])   returns NSTR   division to SCALE places
      $f->fneg() return NSTR                  negation
      $f->fabs() return NSTR                  absolute value
      $f->fcmp(NSTR) return CODE              compare undef,<0,=0,>0
      $f->fround(SCALE) return NSTR           round to SCALE digits
      $f->ffround(SCALE) return NSTR          round at SCALEth place
      $f->fnorm() return (NSTR)               normalize
      $f->fsqrt([SCALE]) return NSTR          sqrt to SCALE places

DESCRIPTION
    All basic math operations are overloaded if you declare your big floats as

$float=newMath::BigFloat"2.123123123123123123123123123123123";
```

```
        number format
        canonical strings have the form /[+-]\d+E[+-]\d+/ . Input
        values can have inbedded whitespace.

    Error returns 'NaN'
        An input parameter was "Not a Number" or divide by zero or
        sqrt of negative number.

    Division is computed to
        `max($div_scale,length(dividend)+length(divisor))' digits by
        default. Also used for default sqrt scale.
    BUGS
       The current version of this module is a preliminary version of
       the real thing that is currently (as of perl5.002) under
       development.

    AUTHOR
       Mark Biggar
```

14.5.5 将 pod 文档转译为 HTML

若要创建一个 HTML 文档，用户可将 pod2html 命令拷贝到自己的目录，并输入：

```
$ pod2html BigFloat.pm > BigFloat.pm.html
```

pod2html 翻译器会创建一个 BigFloat.pm.html 文件。现在请打开浏览器，并在 URL 位置栏中用某种文件协议打开 BigFloat.pm.html；譬如，<file:/yourdirectory path/BigFloat.pm.html>。⊖

14.6 使用 Perl 库中的对象

在第 12 章“模块化、打包并发送到库”中，本书介绍了 Perl 5.6 提供的标准 Perl 库。这些库中含有许多 .pl 和 .pm 文件。在使用这些库时，并不要求用户了解有关 Perl 对象使用方面的知识。它们用到了传统的子例程，而不是方法。现在用户已经了解了如何在 Perl 中使用对象和方法，下面的示例将展示如何使用这些用到 OOP 方法的模块。

14.6.1 另眼看标准 Perl 库

@INC 数组中含有 Perl 会搜索的路径名。在查看了库列表之后，我们再进入（cd）这些标准 Perl 库，看看其中的文件列表。读者会发现，其中有的文件以 pm 扩展名结尾，而有的则以 .pl 扩展名结尾。Perl 5 引入了使用对象的模块文件（以 .pm 结尾），这些模块都能支持 OOP。那些不含扩展名的文件则是一些目录，Perl 通过这些目录来分门别类地保存各类模块。例如，File 和 Math 子目录中分别含有属于各自类别的模块。

⊖ 如果观察到含糊的诊断信息，可能是因为 .pm 文档内容中含有 pod 过滤器无法解析的指向其他页的链接。

示例 14.27

```
1   $ perl -e "print join qq/\n/,@INC;"
    c:/Perl/lib
    c:/Perl/site/lib

2   $ ls /Perl/lib
AnyDBM_File.pm    Exporter.pm      Symbol.pm         cacheout.pl       newgetopt.pl
AutoLoader.pm     ExtUtils         Sys               charnames.pm      open.pm
AutoSplit.pm      Fatal.pm         Term              chat2.pl          open2.pl
B                 Fcntl.pm         Test              complete.pl       open3.pl
B.pm              File             Test.pm           constant.pm       ops.pm
Benchmark.pm      FileCache.pm     Text              ctime.pl          overload.pm
ByteLoader.pm     FileHandle.pm    Thread            diagnostics.pm    perl5db.pl
CGI               FindBin.pm       Thread.pm         dotsh.pl          perllocal.pod
CGI.pm            Getopt           Tie               dumpvar.pl        pwd.pl
CORE              I18N             Time              exceptions.pl     re.pm
CPAN              IO               UNIVERSAL.pm      fastcwd.pl        shellwords.pl
CPAN.pm           IO.pm            User              fields.pm         sigtrap.pm
Carp              IPC              Win32.pod         filetest.pm       stat.pl
Carp.pm           Math             XSLoader.pm       find.pl           strict.pm
Class             Net              abbrev.pl         finddepth.pl      subs.pm
Config.pm         O.pm             assert.pl         flush.pl          syslog.pl
Cwd.pm            Opcode.pm        attributes.pm     ftp.pl            tainted.pl
DB.pm             POSIX.pm         attrs.pm          getcwd.pl         termcap.pl
Data              POSIX.pod        auto              getopt.pl         timelocal.pl
Devel             Pod              autouse.pm        getopts.pl        unicode
DirHandle.pm      SDBM_File.pm     base.pm           hostname.pl       utf8.pm
Dumpvalue.pm      Safe.pm          bigfloat.pl       importenv.pl      utf8_heavy.pl
DynaLoader.pm     Search           bigint.pl         integer.pm        validate.pl
English.pm        SelectSaver.pm   bigrat.pl         less.pm           vars.pm
Env.pm            SelfLoader.pm    blib.pm           lib.pm            warnings
Errno.pm          Shell.pm         bytes.pm          locale.pm         warnings.pm
Exporter          Socket.pm        bytes_heavy.pl    look.pl


3   $ cd Math
4   $ ls
    BigFloat.pm   BigInt.pm    Complex.pm    Trig.pm
```

解释

1. 打印 @INC 数组的元素，确保标准 Perl 库包含在 Perl 的库搜索路径中。
2. 列出所有库例程。
3. 不含 .pl 或 .pm 扩展名的文件是子目录。切换到 Math 子目录中。
4. 列出目录的内容，这里显示了三个 Math 模块。

14.6.2 一个来自标准 Perl 库的面向对象模块

下面这个 BigFloat.pm 模块允许用户使用固定长度的浮点数。数字字符串的格式必须符合 /[+-]\d*\.?\d*E[+-]\d+/。如果返回的是 NaN，就意味着用户输入的内容不是数字，或者除数为 0，又或者是对负数求平方根。BigFloat 用到了 overload 模块，允许用户将方法逻辑赋予 Perl 内建的运算符，从而使运算符产生新的行为方式。在该过程中，运算符相当于一个键，而赋给它的方法则是其对应的值。（详见标准 Perl 中的 overload.pm。）

示例 14.28

```
(The File: BigFloat.pm)

1   package Math::BigFloat;
2   use Math::BigInt;

    use Exporter;  # Just for use to be happy
    @ISA = (Exporter);

3   use overload
4   '+'    =>  sub {new Math::BigFloat &fadd},
    '-'    =>  sub {new Math::BigFloat
                $_[2]? fsub($_[1],${$_[0]}) : fsub(${$_[0]},$_[1])},
    '<=>'  =>  sub {new Math::BigFloat
                $_[2]? fcmp($_[1],${$_[0]}) : fcmp(${$_[0]},$_[1])},
    'cmp'  =>  sub {new Math::BigFloat
                $_[2]? ($_[1] cmp ${$_[0]}) : (${$_[0]} cmp $_[1])},
    '*'    =>  sub {new Math::BigFloat &fmul},
    '/'    =>  sub {new Math::BigFloat
                $_[2]? scalar fdiv($_[1],${$_[0]}) :
                   scalar fdiv(${$_[0]},$_[1])},
    'neg'  =>  sub {new Math::BigFloat &fneg},
    'abs'  =>  sub {new Math::BigFloat &fabs},
    qw(
        ""   stringify
        0+   numify)   # Order of arguments unsignificant
        ;

5   sub new {
       my ($class) = shift;
       my ($foo) = fnorm(shift);
6      panic("Not a number initialized to Math::BigFloat")
              if $foo eq "NaN";
7      bless \$foo, $class;
    }

     < Methods continue here. Module was too long to put here>

    # addition
8   sub fadd { #(fnum_str, fnum_str) return fnum_str
       local($x,$y) = (fnorm($_[$[]),fnorm($_[$[+1]));
       if ($x eq 'NaN' || $y eq 'NaN') {
          NaN';
       } else {
           local($xm,$xe) = split('E',$x);
           local($ym,$ye) = split('E',$y);
           ($xm,$x e,$ym,$ye) = ($ym,$ye,$xm,$xe) if ($xe < $ye);
           &norm(Math::BigInt::badd($ym,$xm.('0' x ($xe-$ye))),$ye);
       }
    }

     < Methods continue here>

    # divisionbb
    # args are dividend, divisor, scale (optional)
    # result has at most max(scale, length(dividend),
```

```
   # length(divisor)) digits

9  sub fdiv    #(fnum_str, fnum_str[,scale]) return fnum_str
   {
      local($x,$y,$scale) = (fnorm($_[$[]),
                             fnorm($_[$[+1]),$_[$[+2]);
      if ($x eq 'NaN' || $y eq 'NaN' || $y eq '+0E+0') {
          'NaN';
      } else {
         local($xm,$xe) = split('E',$x);
         local($ym,$ye) = split('E',$y);
         $scale = $div_scale if (!$scale);
         $scale = length($xm)-1 if (length($xm)-1 > $scale);
         $scale = length($ym)-1 if (length($ym)-1 > $scale);
         $scale = $scale + length($ym) - length($xm);
         &norm(&round(Math::BigInt::bdiv($xm.('0' x $scale),$ym),
                                 $ym),$xe-$ye-$scale);
      }
   }
```

解释

1. 声明 BigFloat 类。它位于标准 Perl 库的 Math 子目录中。
2. BigFloat 类也需要用到 BigFloat.pm 模块。
3. overload 函数允许用户改变 Perl 内建运算符的涵义。例如，在使用 BigFloat.pm 时，+ 运算符相当于键，而负责创建一个对象并调用 &fadd 的匿名子例程则相当于其值。
4. 重载（overload）+ 运算符。见上面的说明内容。
5. 这是 BigFloat 类的构造函数方法，用于创建对象。
6. 如果不是数字，则打印本行信息。
7. 把对象归入到类中。
8. 这是在对象上执行加法运算的子例程。
9. 这是在对象上执行除法运算的子例程。

14.6.3 使用标准 Perl 库中的模块

示例 14.29

```
#!/bin/perl
use Math::BigFloat;  # BigFloat.pm is in the Math directory

$number = "000.95671234e-21";
$mathref = new Math::BigFloat("$number");  # Create the object

print "\$mathref is in class ", ref($mathref), "\n";
                  # Where is the object

print $mathref->fnorm(), "\n";  # Use methods from the class

print "The sum of $mathref + 500 is: ", $mathref->fadd("500"),
                                 "\n";
print "Division using overloaded operator: ", $mathref / 200.5,
                                 "\n";
print "Division using fdiv method:", $mathref->fdiv("200.5"),
                                 "\n";
```

```
10  print "Enter a number ";
    chop($numstr = <STDIN>);

11  if ( $mathref->fadd($numstr) eq "NaN" ){
                    print "You didn't enter a number.\n"};

# Return value of NaN means the string is not a number,
# or you divided by zero, or you took the square root
# of a negative number.

(Output)
5   $mathref is in class Math::BigFloat
6   +95671234E-29
7   The sum of .000000000000000000000095671234 + 500 is:
    +50000000000000000000000095671234E-29
8   Division using overloaded operator:
    .000000000000000000000004771632618453865336658354114713216957606
9   Division using fdiv method:
    +4771632618453865336658354114713216957606E-63
10  Enter a number hello
11  You didn't enter a number.
```

解释

1. 这是 Perl 解释器的 shbang 行。
2. use 函数将模块 BigFloat 加载到程序中。由于该模块位于 Math 库的子目录下，因此可通过在模块名前面加上子目录名和双冒号的方式来导入它。
3. 将 $number 赋值为一个大数（e 标记）。
4. 现在使用模块中的方法。调用 BigFloat 的构造函数。返回指向对象的引用，并将它赋予 $mathref。
5. ref 函数负责返回类名。
6. fnorm 方法以带符号的科学计数法形式返回 $number 的“常规”值。
7. fadd 方法给该数字加上 500。
8. 在本例中用到了一个重载的运算符。本行把类方法 fdiv 赋予了 / 运算符，并用它来执行除法运算。参见上面的 BigFloat.pm 模块代码。
9. 这一次不使用重载，而是直接调用 fdiv 方法来执行除法运算。其输出内容稍有不同。
10. 要求用户输入一个数字。
11. 如果 fadd 方法返回 NaN（非数字），则打印本行信息。这是一种检查用户输入数字有效性的方法。

14.7 读者应当学到的内容

1. OOP 是什么意思？
2. 包（package）和类（class）有什么区别？
3. 什么是方法（method）？
4. 类方法接受的一个参数是什么？
5. 什么函数能够创建对象？

6. 什么是属性（property）？
7. 什么是实例方法（instance method）？
8. Perl 提供“private”关键字吗？
9. 如何命名一个类？应当把类放在哪里？
10. 类方法调用是什么意思？
11. 什么是多态性（polymorphism）？
12. @ISA 数组有什么用途？
13. 什么是派生类（derived class）？
14. 闭包（closure）是做什么用的？
15. 如何为类提供文档？
16. 什么是 pod 过滤器？
17. 如何使用 pod 指令？

14.8 下章简介

在下一章里，读者将学习 Perl 内建的一种能把变量绑到类里的预定义方法，并学习如何通过该函数去使用 DBM，即用于存储大量二进制文件的数据库管理库。

练习 14 本课的对象是什么

第一部分 对象入门

1. 编写一个名为 Rightnow.pm 的模块，含有如下三个组件：
   a. 一个名叫“new”的构造函数。
   b. 一个用于设置时间的“set_time”方法。该方法需要用到 localtime 函数。
   c. 一个负责打印时间的“print_time”方法。
   该方法将根据参数决定是用Military格式还是用标准格式来打印时间；譬如，print_time("Military");
   d. 在另一个程序中，使用 Rightnow 模块创建一个 Rightnow 对象，并调用其“print_time”方法，打印如下内容：
   Time now: 2:48:20 PM
   Time now: 14:48:20

第二部分 对象深入

1. 在类中创建一个 Student 对象，并把 Student 对象的属性作为参数传递给构造函数方法。Student 对象中含有三个属性：学生姓名、学生专业、以及该学生选修的课程列表。
   创建一个名叫“show_student”的实例方法，负责打印 Student 对象内容。
   模块调用代码将创建两个 Student 对象，并显示每个对象的内容。
2. 向 Student 对象添加四个新属性：学生地址、学生 ID 号、入学日期以及学费。例如：

   ID: 123A
   StartDate: 01/10/07
   Tuition: 5400.55

应当如何管理这些属性呢？如果需要向构造函数传递这么多参数的话，用户还不如另外创建一个名叫“set_student”的实例方法。

创建三个新的 Student 对象。

3. 创建两个新的带参数的访问方法。其中一个名叫“add_course”，另一个则是“drop_course”。这两个用户接口允许用户添加或去除任意数目的课程信息，只需把一组参数传递给相应的方法即可。譬如，$ptr->add_course(["C++","Java"]);
4. 这里将使用一个变量“class”来跟踪新学生的数量。每当添加一个学生时，就刷新这个计数器。在退出程序前，打印新学生总数。这里应当使用 DESTROY 方法。
5. 从这里开始，把每个学生的信息数据发送至一个文件。该文件内容应遵循如下格式：

```
John Doe:14 Main St:3456IX:Math:Trigonometry, Calculus, French:01/01/06:4500
```

6. 创建另一个文件，负责跟踪学生总数。每当启动脚本时，就从该文件中读取学生总数。每当添加新学生时，就对他说：“Welcome, John D.”

第三部分　创建一个面向对象的模块

1. 将 Checking.pm 模块转换为面向对象的形式。其对象名叫“the balance”，其子例程则是“methods”。其构造函数将至少含有两个属性：余额和账号。其中，账号将作为参数传递给构造函数。而余额则是从登记表中检索得到的，其初始值为 0。

   当创建登记表文件时，须把账号追加到文件名上。登记表文件提供了账号、余额和日期信息。

   这里可使用之前在 ATM 用户脚本中已经创建好的 Checking.pm 模块。
2. 读者可否做到创建多个 Checking 对象，并为每个账户分别维护其余额信息？

第四部分　使用 @ISA 和继承机制

1. 创建一个带有构造函数和单个访问方法的 Pet 类。其构造函数提供了一些涉及普通宠物的属性，如：

   主人
   名字
   性别

   a. 其访问方法名叫 eat()。该方法接受一个参数：某种宠物食用的食物种类。例如，狗吃 Alpo。Dog 类自己不提供 eat() 方法，而是从该类继承。

   b. 创建两个继承 Pet 类的类; 譬如，一个 Dog 类和一个 Cat 类。它们都将用到 Pet 类的构造函数，并添加一些自己独有的属性。它们将提供 speak() 方法，但不提供自己的 eat() 方法。
2. 这里我们将创建一个名叫 Bank.pm 的基类，并另外创建两个使用它的模块：Checking.pm 和 Savings.pm。

   a. Bank.pm 父类的构造函数是可有可无的，但是它必须为 Checking.pm 提供 deposit()、withdraw() 和 get_balance() 方法。

   b. 从 Checking.pm 中移除 deposit() 和 withdraw()。使用 Checking.pm 的程序必须通过 @ISA 数组从 Bank.pm 中继承这两个方法。

   c. 创建另一个名叫 Savings.pm 的模块。

   d. Checking.pm 和 Savings.pm 都将用到 Bank 模块，并继承其方法。这两个模块都拥有自己的构造函数和属性。其中一个属性是账户状态。它可以取“active”或者“closed”。

   Savings 账户每天会收取 1% 的累积利息，并要求初始账户余额至少为 $200。

   Checking 账户则有透支保护功能，并会为每笔交易收取 $35 的费用。它不允许透支超过 $300，

并要求开户余额不少于 $25。

e. Checking.pm 和 Savings.pm 模块将拥有各自独立的账号和登记文件。

f. ATM 脚本将用到上面这两个模块。用户脚本会提供一个菜单，允许用户选择两个账户中的任意一个。在得到一个新的账户对象后，用户还可为账户选择交易类型（位于原有 Checking.pm 模块的子菜单中），并继续进行交易，直到退出交易为止。当用户退出交易时，其账户登记文件将自动刷新，并提示用户是否需要返回主菜单。如果用户回答“是”，则他就能再次看到主菜单；如果用户回答“否”，则退出整个程序。用户必须为各账户的登记文件提供惟一的名字，以方便区分 savings 和 checking 这两个账户。

示例：

```
perl user.pl
        Welcome!
Select an account type:
  1) Checking
  2) Savings
1
Select a function:
  1) deposit
  2) withdraw
  3) get balance
  4) exit
1
How much do you want to deposit? 5
Select a function:
  1) deposit
  2) withdraw
  3) get balance
  4) exit
3
Your balance is $30.00
Select a function:
  1) deposit
  2) withdraw
  3) get balance
  4) exit
2
perl user.pl
        Welcome!
Select an account type:
  1) Checking
  2) Savings
1
Select a function:
  1) deposit
  2) withdraw
  3) get balance
  4) exit
1
How much do you want to deposit? 5
Select a function:
  1) deposit
  2) withdraw
  3) get balance
```

```
    4) exit
  3
  Your balance is $30.00
  Select a function:
    1) deposit
    2) withdraw
    3) get balance
    4) exit
  2
  Return to the main menu? y
         Welcome!
  Select an account type:
    1) Checking
    2) Savings
  2
  Select a function:
    1) deposit
    2) withdraw
    3) get balance
    4) exit
  3
  Your balance is $100.00
  Select a function:
    1) deposit
    2) withdraw
    3) get balance
    4) exit
  1
  How much do you want to deposit? 25
  Select a function:
    1) deposit
    2) withdraw
    3) get balance
    4) exit
  4
  After interest balance is 127.50
```

第五部分

进入标准 Perl 库中的 pod 目录，查看 perlpod.html。该文件中含有 Larry Wall 提供的有关如何使用 pod 命令归档 Perl 程序的用户接口。

进入浏览器，在地址栏中键入：

```
file:/<directory-to-your-library-file>/Pod/pod.html
```

便可看到有关创建 pod 文档的指令。

然后为 Checking.pm 模块创建一个公共接口。在 Checking.pm 脚本中嵌入 pod 命令，以便向用户解释如何使用这个模块。在此过程中，请读者遵守库模块的相关指导；譬如：必须提供 NAME、SYNOPIS、DESCRIPTION 和 AUTHOR 等。通过 pod2html 过滤器来执行 pod 文件，并在浏览器中显示文档内容。读者也可使用 perldoc 命令将文档内容打印到终端屏幕上。

第 15 章　神奇的 Tie 和 DBS

15.1　连接变量与类

在通常情况下，当对变量执行某种操作如赋值、修改或打印变量内容时，Perl 会在内部对该变量进行一些必要的处理。譬如，在创建变量并为它赋值时，用户无需使用构造函数方法；而在操纵变量时，也没有必要为它创建访问方法。赋值语句 $x=5 并没有提供过于复杂的语义。Perl 会为 $x 创建一个内存位置，并将值 5 放到该位置中。

现在，用户可以把常规变量和类连接到一起，以便为该变量提供方法，从而能在赋值或检索变量值时变换相应的变量，就好像变魔术一样。然后，用户便可为这些标量、数组或散列提供新的具体值。和那些必须通过引用来访问的对象不同，连接变量在创建后就和其他变量一视同仁。所有这些细节都将对用户隐藏。用户可以通过与连接前一样的语法给变量赋值或访问这些变量[⊖]。其神奇之处在于幕后的动作。Perl 会创建一个负责表达该变量的对象，并使用预定义的方法名构造、设置、获取和撤销连接到变量的对象。创建类的程序员可使用预定义的方法名（如 FETCH 和 STORE）来导入操作对象时需要使用的语句。用户能连接一个变量，从此以后，就可以在程序中像使用其他变量一样地使用它。

15.1.1　tie 函数

tie 函数负责把变量连接到一个指定的包或类上，并返回一个指向对象的引用。其所有细节都在内部予以处理。tie 函数常常会和关联数组配合使用，以便将其键 / 值对连接到数据库上；譬如，在 Perl 发布包提供的 DBM 模块中使用连接变量。untie 函数负责断开变量与类之间的连接。tie 函数的格式如下：

格式

```
$object =  tie variable, class, list;
untie variable;

tie variable, class, list;
$object = tied variable;
```

⊖　DBM 数据库借助 tie 机制自动对数据执行数据库操作。

tie 函数会返回指向一个之前已经绑定 tie 函数的对象的引用。如果该变量没有与包连接，则返回一个未定义的引用。

15.1.2 预定义方法

变量连接机制允许用户通过构造具有特殊方法的类来定义变量的行为，这些特殊方法可用于创建和访问变量。在使用变量时，将自动调用这些方法。变量并不是与任何类都能连接的，而是必须连接到含有某个预定义方法名的类。变量的行为将由使用该变量时自动调用的方法来决定。用于连接（构造函数）和操纵（访问方法）的构造函数和方法都拥有预先定义好的名称。程序会在读取、存储或撤销连接变量时自动调用这些方法。其所有的细节都会在内部予以处理。构造函数可以请求并返回指向任意类型对象的指针。例如，引用可以指向请求的标量、数组或散列。但是，如果使用了 TIESCALAR 的话，则访问方法必须返回标量值；如果使用的是 TIEARRAY，则应返回数组；如果使用的是 TIEHASH，则应返回散列。

15.1.3 连接标量

为了使用连接的标量，必须为类定义一组具有预定义名称的方法。在连接标量时，应当调用构造函数 TIESCALAR，后者会创建一个底层对象，并通过访问方法 STORE 和 FETCH 来操纵它。在给连接的标量赋值时，用户应当调用 STORE 方法；如果要显示连接变量的内容，则应调用 FETCH 方法。这里不要求提供 DESTROY 方法。不过如果已经定义了这个方法，程序就会在取消标量连接或者在超出作用域时调用它。针对连接标量提供的方法有如下几种：

```
TIESCALAR $classname, LIST
STORE $self, $value
FETCH $self
DESTROY $self
```

在标准 Perl 库中还提供了一种基本模块，用于连接那些为标量连接类提供骨架方法的标量。如要获得在连接标量与包时可供使用的函数列表，请参阅 perltie man 页面。基本的 Tie::Scalar 包提供了 new 方法，也提供了 TIESCALAR、FETCH 和 STORE 方法。如需了解该模块，请键入 perldoc Tie::Scalar。

示例 15.1 展示了如何连接一个标量，以及如何访问连接后的标量。

示例 15.1

```
# File is Square.pm
# It will square a number, initially set to 5
package Square;
sub TIESCALAR{
    my $class = shift;
    my $data = shift;
    bless(\$data,$class);   # Blessing a scalar
}
sub FETCH{
    my $self = shift;
    $$self **= 2;
}
```

```
9   sub STORE{
10      my $self = shift;
11      $$self = shift;
    }
    1;
--------------------------------------------------------------------

    # User program
12  use Square;
13  $object=tie $squared, 'Square', 5;  # Call constructor TIESCALAR
14  print "object is $object.\n";
15  print $squared,"\n";      # Call FETCH three times
16  print $squared,"\n";
    print $squared,"\n";
    print "----------------------------\n";

17  $squared=3;            # Call STORE

18  print $squared,"\n";   # Call FETCH
    print $squared,"\n";
    print $squared,"\n";

19  untie $squared;      # Break the tie that binds the
                         # scalar to the object
(Output)
14  object is Square=SCALAR(0x1a72da8).
15  25
16  625
    390625
    ----------------------------
18  9
    81
    6561
```

解释

1. 声明包 / 类 Square。其文件是 Square.pm。
2. 类构造函数是 TIESCALAR。它会在待连接标量和对象之间创建关联关系。参见本示例的第 13 行。在这里调用构造函数。连接到对象的变量是 $squared。
3. 构造函数的第一个参数是类名。它由 @_ 数组移出，并赋值给 $class。
4. 再看看第 13 行。连接变量 $squared 的后面跟随着类名，其后则是数字 5。在构造函数中，从数组 @_ 移出数字 5，并赋值给 $data。
5. 创建指向标量的引用，并传递给 bless 函数。该函数负责创建对象。
6. 定义 FETCH 方法。该访问方法负责从对象中检索数据。
7. FETCH 方法的第一个参数是指向对象的一个引用。
8. 按地址访问指针，并对它指向的内容计算平方值。
9. 定义 STORE 方法。该方法用于给对象赋值。
10. STORE 方法的第一个参数是指向对象的一个引用。
11. 把需要赋予的值移出 @_ 数组，并赋值给 $self。
12. 这是用户程序 / 驱动程序。负责把 Square 模块加载到程序中。
13. tie 函数能自动调用 TIESCALAR 构造函数。tie 函数能把变量 $squared 连接到类上，并返回指向新建对象的引用。

14. 引用 $object 指向在 Square 类中找到的标量型变量。
15. 自动调用 FETCH 方法，打印 $squared 变量的当前值。
16. 再次调用 FETCH 方法。每当调用该方法时，程序就会再次计算连接变量的平方值，并返回计算结果。
17. 自动调用 STORE 方法。该方法会把连接变量赋值为 3。
18. 自动调用 FETCH 方法，显示其平方值。
19. untie 方法能解除对象和连接变量之间的关联。如果定义了 DESTROY 方法，则在这里将自动调用它。

15.1.4 连接数组

如要使用连接数组，必须在类中定义一组具有预定义名称的方法。在连接变量时，程序会调用构造函数 TIEARRAY，由它负责创建一个底层对象，并通过访问方法 STORE 和 FETCH 操纵该对象。当需要为连接的数组赋值时，用户应调用 STORE 方法；如需显示连接数组的内容，则应调用 FETCH 方法。这里不强制要求提供 DESTROY 方法。不过如果已经定义了这个方法，程序就会在取消数组连接或者在超出作用域时调用它。还有其他一组可选的方法可用于连接数组。其中，STORESIZE 方法负责设置数组中的项目总数。FETCHSIZE 方法可以通过 scalar(@array) 或 $#array+1 获取数组的大小。而在清空数组所有各项时，可以使用 CLEAR 方法。此外还有许多其他的方法，譬如 POP 和 PUSH，这两个方法在处理数组时的功能与同名的 Perl 函数相同。为数组提供的方法包括如下几种：

```
TIEARRAY $classname, LIST
STORE $self, $subscript, $value
FETCH $self, $subscript, $value
DESTROY $self
STORESIZE  $self, $arraysize
FETCHSIZE  $self
EXTEND $self, $arraysize
EXISTS $subscript
DELETE $self, $subscript
CLEAR $self
PUSH $self, LIST
UNSHIFT $self, LIST
POP $self
SHIFT $self
SPLICE $self, OFFSET, LENGTH, LIST
```

此外，标准 Perl 库中还提供了一个名叫 Tie::Array 的类，该类含有许多上面列出的预定义方法，从而令数组连接过程更为方便。如要查看该模块的文档，请键入 perldoc Tie::Array。

示例 15.2

```
package Temp;
sub TIEARRAY {
    my $class = shift;  # Shifting the @_ array
    my $obj = [ ];
    bless ($obj, $class);
}
```

```
# Access methods
sub FETCH {
    my $self=shift;
    my $indx = shift;
    return $self->[$indx];
}

sub STORE {
    my $self = shift;
    my $indx= shift;
    my $F = shift;    # The Fahrenheit temperature
    $self->[$indx]=($F - 32) / 1.8;   # Magic works here!

}
1;
---------------------------------------------------------------------
#!/bin/perl
# The user/driver program
use Temp;
tie @list, "Temp";
print "Beginning Fahrenheit: ";
chomp($bf = <STDIN>);
print "Ending temp: ";
chomp($ef = <STDIN>);
print "Increment value: ";
chomp($ic = <STDIN>);
print "\n";
print "\tConversion Table\n";
print "\t----------------\n";

for($i=$bf;$i<=$ef;$i+=$ic){
    $list[$i]=$i;
    printf"\t$i F. = %.2f C.\n", $list[$i]; }


(Output)
Beginning Fahrenheit: 32
Ending temp: 100
Increment value: 5

Conversion Table
----------------
2 F. = 0.00 C.
37 F. = 2.78 C.
42 F. = 5.56 C.
47 F. = 8.33 C.
52 F. = 11.11 C.
57 F. = 13.89 C.
62 F. = 16.67 C.
67 F. = 19.44 C.
72 F. = 22.22 C.
77 F. = 25.00 C.
82 F. = 27.78 C.
87 F. = 30.56 C.
92 F. = 33.33 C.
97 F. = 36.11 C.
```

解释

1. 这是一个包 / 类的声明。文件是 Temp.pm。
2. TIEARRAY 是连接数组的构造函数。它负责创建底层对象。
3. 传递给构造函数的第一个参数是类名 Temp。
4. 创建一个指向匿名数组的引用。
5. 把对象归入到类中。
6. 访问方法 FETCH 负责从已连接的数组中检索数据元素。
7. 移出第一个参数，即指向对象的引用，并将其赋值给 $self。
8. 下一个参数是连接数组中下标的值。移出它并赋值给 $indx。
9. 当用户程序 / 驱动程序试图显示已连接数组中的一个元素值时，返回数组元素的值。
10. 访问方法 STORE 负责为已连接的数组赋值。
11. 移出第一个参数，即指向对象的引用，并将其赋值给 $self。
12. 下一个参数是连接数组中下标的值。移出它并赋值给 $indx。
13. 从参数列表中移出华氏温度值。参见第 19 行，即在用户程序 / 驱动程序中完成赋值操作。在 STORE 方法内，要赋的值将保存在 $F 中。
14. 对于接受到的已连接数组元素，执行从华氏度到摄氏度的转换运算。用户程序 / 驱动程序将始终看不到该计算过程，就好像是在变魔术一样。
15. 这是用户提供的程序，加载了 Temp.pm 模块。
16. tie 函数负责把数组连接到类 Temp 上，并返回指向一个连接到数组上的底层对象的引用。
17. 要求用户提供输入。用户将提供起始的华氏温度值、最后的华氏温度值、以及中间的增量值。
18. 进入 for 循环，迭代遍历温度列表。在转换完毕后，绘制华氏温度对应到摄氏温度的图表。
19. 奇妙的地方出现在这里。在赋值时，程序会自动调用 STORE 方法，利用公式完成从华氏温度值到摄氏温度值的转换，并将结果赋值给数组。这里需要把指向已连接数组的引用和数组下标一同传递给 STORE 方法。
20. 在使用 printf 函数时，自动调用 FETCH 方法，并显示已连接数组中的元素值。这里需要把指向已连接数组的引用和数组下标一同传递给 FETCH 方法。

15.1.5 连接散列

如要连接散列，必须在类中定义一组具有预定义名称的方法。在连接变量时，程序会调用构造函数 TIEHASH 创建一个底层对象，并通过访问方法 STORE 和 FETCH 操纵该对象。当需要为连接的散列赋值时，应调用 STORE 方法；如需显示连接散列的内容，则应调用 FETCH 方法。这里不强制要求提供 DESTROY 方法。不过如果已经定义了这个方法，程序就会在取消散列连接或在超出作用域时调用它。还有其他一组可选的方法可用于连接散列。其中，DELETE 方法能够移除指定的键 / 值对；EXISTS 方法负责检查某一个键是否存在；而 CLEAR 方法则负责清空整个散列。如果用户使用了 Perl 内建的 keys、values 或 each 方法，则可调用 FIRSTKEY 和 NEXTKEY 来遍历整个散列。与连接散列有关的方法包括如下几种：

```
TIEHASH $classname, LIST
FETCH $self, $key
STORE $self, $key
DELETE $self, $key
```

```
EXISTS $self, $key
FIRSTKEY $self
NEXTKEY $self, $lastkey
DESTROY $self
CLEAR $self
```

此外，标准 Perl 库中还提供了一个名叫 Tie::Hash 的类，该类含有许多上面列出的预定义方法，从而令散列连接过程变得更为方便。如要查看该模块文档，请键入 perldoc Tie::Hash。

示例 15.3

```
  (The Script)
     #!/bin/perl
     # Example using tie with a hash

1    package House;
2    sub TIEHASH {               # Constructor method
3        my $class = shift;      # Shifting the @_ array
         my $price = shift;
         my $color = shift;
         my $rooms = shift;
4        print "I'm the constructor in class $class.\n";
5        my $house =  {  Color=>$color,     # Data for the tied hash
6                        Price=>$price,
                         Rooms=>$rooms,

                      };
7        bless $house, $class;
     }
8    sub FETCH {                 # Access methods
         my $self=shift;
         my $key=shift;
9        print "Fetching a value.\n";
10       return $self->{$key};
     }
11   sub STORE {
         my $self = shift;
         my $key = shift;
         my $value = shift;
         print "Storing a value.\n";
12       $self->{$key}=$value;
     }
     1;

---------------------------------------------------------------------

     # User/driver program
13   use House;
     # The arguments following the package name are
     # are passed as a list to the tied hash
     # Usage: tie hash, package, argument list
     # The hash %home is tied to the package House.
14   tie %home, "House", 155000, "Yellow", 9;
                    # Calls the TIEHASH constructor
15    print qq/The original color of the house: $home{"Color"}\n/;
                    # Calls FETCH method
```

```
16  print qq/The number of rooms in the house: $home{"Rooms"}\n/;
17  print qq/The price of the house is: $home{"Price"}\n/;
18  $home{"Color"}="beige with white trim";   # Calls STORE method
19  print "The house has been painted. It is now $home{Color}.\n";
20  untie(%home);      # Removes the object
(Output)
4   I'm the constructor in class House.
9   Fetching a value.
15  The original color of the house: Yellow
9   Fetching a value.
16  The number of rooms in the house: 9
9   Fetching a value.
17  The price of the house is: 155000
    Storing a value.
    Fetching a value.
    The house has been painted. It is now beige with white trim.
```

解释

1. 声明包 / 类 House。文件是 House.pm。
2. 构造函数 TIEHASH 负责把散列连接到对象。
3. 第一个参数是类名 $class。将其他参数移出数组，并赋值为匿名散列（即对象）中各个键的对应值。
4. 打印输出内容，表明构造函数类名为 House。
5. 创建一个指向匿名散列的引用，并为其中的键 / 值对赋值。它们将成为对象中的属性。
6. 把赋予匿名散列的值传递给构造函数 TIEHASH（详见用户程序的第 13 行）。
7. bless 函数将返回一个指向新建对象的引用。
8. FETCH 是一个访问方法，负责检索散列对象中的值。
9. 每当用户使用 print 函数显示散列中的值时，程序便会自动调用 FETCH 方法，并打印本行内容。
10. 返回来自散列的值。
11. 当用户需要给散列中的每个键赋值时，程序便会自动调用访问方法 STORE。
12. 要赋值给散列的值来自于传递到 STORE 方法的参数列表。
13. 这是一段用户程序 / 驱动程序。把 House 模块加载到程序中。
14. 在 House.pm 模块中利用 tie 函数调用 TIEHASH 构造函数。散列 %home 将会连接到 House 类中的对象上。传递类名 House 及其三个附加参数。
15. 调用 FETCH 访问方法，显示散列中对应于 Color 键的值。
16. 调用 FETCH 访问方法，显示散列中对应于 Rooms 键的值。
17. 调用 FETCH 访问方法，显示散列中对应于 Price 键的值。
18. 在为散列赋值时，程序会自动调用 STORE 访问方法。在这里是 Color 键。
19. print 函数将导致程序自动调用 FETCH 访问方法显示指定散列键（Color）的值。
20. untie 函数负责解除散列和所连接对象之间的关联。

示例 15.4

```
    # File is House.pm
1   package House;
2   sub TIEHASH  {
        my $class = shift;
        print "I'm the constructor in package $class\n";
```

```
        my $houseref = {};
        bless $houseref, $class;
    }
3   sub FETCH {
        my $self=shift;
        my $key=shift;
        return $self->{$key};
    }
4   sub STORE {
        my $self = shift;
        my $key = shift;
        my $value = shift;
        $self->{$key}=$value;
    }

5   sub FIRSTKEY {
        my $self = shift;
6       my $tmp = scalar keys %{$self};
7       return each  %{$self};
    }
8   sub NEXTKEY {
        $self=shift;
        each %{$self};
    }
    1;

 --------------------------------------------------------------------

      #!/usr/bin/perl
      # File is mainfile
9   use House;
10  tie %home, "House";
    $home{"Price"} = 55000;  # Assign and Store the data
    $home{"Rooms"} = 11;
    # Fetch the data
    print "The number of rooms in the house: $home{Rooms}\n";
    print "The price of the house is: $home{Price}\n";
11  foreach $key (keys(%home)){
12      print "Key is $key\n";
    }
13  while(  ($key, $value) = each(%home)){
            # Calls to FIRSTKEY and NEXTKEY
14      print "Key=$key, Value=$value\n";
    }
15  untie(%home);

(Output)
I'm the constructor in package House
The number of rooms in the house: 11
The price of the house is: 55000
Key is Rooms
Key is Price
Key=Rooms, Value=11
Key=Price, Value=55000
```

解释

1. 声明包 / 类 House。文件是 House.pm。
2. 构造函数 TIEHASH 负责把散列连接到对象。
3. FETCH 方法是一个访问方法，负责检索散列对象中的值。
4. STORE 方法是一个访问方法，负责将一个值赋予散列对象。

5. 如果用户程序调用了 Perl 内建的 keys、values 或 each 函数的话，程序便会自动调用 FIRSTKEY 方法。
6. 如果在标量上下文语境中调用键，Perl 会重置散列的内部状态，以保证下一次调用 each 函数时仍然能够给出第一个键。
7. each 函数会返回第一个键 / 值对。
8. 当用户程序借助循环迭代遍历某个散列时，NEXTKEY 方法会知道前一个键（PREVKEY）是什么，并从它开始使用下一个键。
9. 这是一段用户程序 / 驱动程序。把 House 模块加载到程序中。
10. tie 函数会在 House.pm 模块中调用 TIEHASH 构造函数。散列 %home 将连接到 House 类中的对象上。
11. while 循环会使用 keys 函数迭代遍历整个散列。在第一次进入循环时，调用 FIRSTKEY 方法。
12. 打印每个键的对应值。这个值是由访问方法 FIRSTKEY 和 NEXTKEY 返回的。
13. while 循环会使用 keys 函数迭代遍历整个散列。在第一次进入循环时，调用 FIRSTKEY 方法。
14. 打印每个键和每个值。这些值都是由访问方法 FIRSTKEY 和 NEXTKEY 返回的。

15.2 DBM 文件

Perl发布包中带有一组称作DBM(即database management的缩写)的数据库管理库文件。DBM文件的概念最早源于早期的 UNIX 系统，它由一组能够随机访问其记录的 C 库程序组成。DBM 数据库文件以键 / 值对的形式保存数据，即可以映射为磁盘文件中的关联数组。DBM 支持多种不同的形态，它们都提供了有关为什么要使用连接散列的最佳原因。

DBM 文件是一种二进制文件，可以处理极为庞大的数据库。使用 DBM 函数来存储数据的优点在于其数据是持久化的；即任何程序通过 DBM 函数都能访问这些文件。其缺点在于，迄今为止它还不能支持复杂数据库结构、索引、多重表等特性，也没有可靠的文件锁定和缓冲区清除机制，从而令并行化读写和更新操作存在潜在的风险㊀。用户可以使用 Perl 的 flock 函数实现文件锁定，但是如何正确配置该操作策略的相关内容超出了本书介绍的范围㊁。

用户不必指明需要使用哪个标准 DBM 包，因为 AnyDBM_File.pm 模块能从标准 Perl 库的标准集合中为系统获得正确的包。如果程序需要运行在多种平台上，AnyDBM_File 模块也是非常有用的。它能为五种不同类型的实现选取正确的包，详见表 15-1：

表 15-1　DBM 的实现

odbm	UNIX 系统中“旧”的 DBM 实现，已由 NDBM 取代
ndbm	UNIX 系统中“新”的 DBM 实现
sdbm	标准 Perl DBM，提供了跨平台兼容性，但是只适用于不大的数据库
gdbm	GNU DBM，是一种快速、可移植的 DBM 实现，详见 www.gnu.org
bsd-db	BSD UNIX 系统中的 Berkley DB 实现；它是所有 DBM 中最强大的，详见 www.sleepycat.com

㊀ 尽管 Perl 的 tie 函数可以用于替代 dbmopen 函数，但这里还是使用后者，因为它比 tie 函数更简单。

㊁ 有关文件锁定的详细内容，请参阅 Descartes, A., and Bunce, T., Programming the Perl DBI, O'Reilly & Associates, 2000, p.35。

下面这些内容来自于 AnyDBM_File 文档，列出了各种 DBM 实现中的一些区别。用户可在命令行提示符下键入：

```
perldoc AnyDBM_File
                            odbm  ndbm    sdbm    gdbm    bsd-db
                            ----  ----    ----    ----    ------
  Linkage comes w/ perl     yes     yes     yes     yes     yes
  Src comes w/ perl         no      no      yes     no      no
  Comes w/ many unix os     yes     yes[0]  no      no      no
  Builds ok on !unix        ?       ?       yes     yes     ?
  Code Size                 ?       ?       small   big     big
  Database Size             ?       ?       small   big?    ok[1]
  Speed                     ?       ?       slow    ok      fast
  FTPable                   no      no      yes     yes     yes
  Easy to build            N/A     N/A      yes     yes     ok[2]
  Size limits               1k      4k      1k[3]   none    none
  Byte-order independent    no      no      no      no      yes
  Licensing restrictions    ?       ?       no      yes     no
```

15.2.1 创建并赋值 DBM 文件

在访问数据库前，必须先使用 dbmopen 函数或 tie 函数打开它。该操作会把 DBM 文件连接到绑定的数组（散列）上。它会创建两个文件：第一个文件含有索引目录，并带有扩展名 .dir；第二个文件则以 .pag 结尾，含有所有的数据。这两个文件都不是可读格式。dbm 函数负责访问数据内容。这些文件对于用户而言都是可见的。

像处理其他任何 Perl 散列一样，将数据赋值给散列。用户可使用 Perl 提供的 delete 函数删除元素。也可使用 dbmclose 或 untie 函数关闭 DBM 文件。

格式

```
dbmopen(hash, dbfilename, mode);
tie(hash, Module , dbfilename, flags, mode);
```

示例 15.5

```
dbmopen(%myhash, "mydbmfile", 0666);
tie(%myhash,SDBM_File, "mydbmfile", O_RDWR|O_CREAT,0640);
```

在从某个 DBM 文件生成为格式化数据时，Perl 提供的报表生成机制便显得非常有用。下面这个示例展示了如何创建、添加、删除和关闭一个 DBM 文件，以及如何生成 Perl 样式的报表。

示例 15.6

```
(The Script)
   #!/usr/bin/perl
    # Program name: makestates.pl
    # This program creates the database using the dbm functions
   use AnyDBM_File;  # Let Perl pick the right dbm for your system
   dbmopen(%states, "statedb", 0666[a]) || die;
                     # Create or open the database
   TRY:  {
       print "Enter the abbreviation for your state. ";
```

```
        chomp($abbrev=<STDIN>);
        $abbrev = uc $abbrev;  # Make sure abbreviation is uppercase
5       print "Enter the name of the state. ";
        chomp($state=<STDIN>);
        lc $state;
6       $states{$abbrev}="\u$state";  # Assign values to the database
7       print "Another entry? ";
        $answer = <STDIN>;
8       redo TRY  if $answer =~ /Y|y/;
    }
9   dbmclose(%states);       # Close the database
----------------------------------------------------------------------

(The Command line)
10  $ ls
    makestates.pl statedb.dir statedb.pag[b]
-------------------------------------------------------------------
(Output)
4   Enter the abbreviation for your state. CA
5   Enter the name of the state. California
7   Another entry? y
    Enter the abbreviation for your state. me
    Enter the name of the state. Maine
    Another entry? y
    Enter the abbreviation for your state. NE
    Enter the name of the state. Nebraska
    Another entry? y
    Enter the abbreviation for your state. tx
    Enter the name of the state. Texas
    Another entry? n
```

a. 在 Win32 系统中会忽略其权限。

b. 在某些版本中，只能创建一个带有 .db 扩展名的文件。

解释

1. AnyDBM_File 模块负责根据具体安装版本来选择合适的 DBM 库。
2. dbmopen 函数负责把一个 DBM 文件与一个散列绑定到一起。在这里，创建的数据库文件名为 statedb，而该散列则名叫 %states。如果该数据库不存在，则必须给出有效的访问权限模式。这里给定的八进制模式为 0666。在 UNIX 型系统中，该模式的意思是允许所有用户对它进行读写操作。
3. 进入标记的代码块。
4. 要求用户输入所在州名的缩写。该输入将用于填充 %states 散列。
5. 要求用户输入所在州名。
6. 把值 state 赋予散列 %states，该散列的键是州名的缩写。\u 转义序列会让州名的第一个字母变为大写形式。在赋值时，程序后台会通过 tie 机制为 DBM 文件赋予新的值。
7. 要求用户把另一项输入到 DBM 文件中。
8. 如果用户还要给 DBM 文件添加其他项，程序便会转到标记为 TRY 的代码块顶部，然后开始处理它。
9. 将 DBM 文件绑定到 %states 散列后，使用 dbmclose 函数中断连接（亦可通过调用 untie 函数来实现）。

10. 清单显示了所有由 dbmopen 函数创建的文件列表。其中第一个文件 makestates.pl 是一个 Perl 脚本。第二个文件 statedb.dir 是一个索引文件。最后一个文件 statedb.pg 则是含有散列数据的文件。

15.2.2　从 DBM 文件中检索数据

在打开 DBM 文件后，Perl 脚本就把它连接到一个相关联的散列上。其所有实现细节都对用户隐藏起来。其数据检索过程既快速又方便。在此过程中，用户可以像对待普通的 Perl 散列一样操作它们。由于散列与 DBM 文件相连，因此检索得到的数据均来自于 DBM 文件。

示例 15.7

```
(The Script)
    #!/bin/perl
    # Program name: getstates.pl
    # This program fetches the data from the database
    # and generates a report

use AnyDBM_File;
dbmopen(%states, "statedb", 0666);   # Open the database
@sortedkeys=sort keys %states;    # Sort the database by keys
    foreach $key ( @sortedkeys ){
    $value=$states{$key};
    $total++;
    write;
}
dbmclose(%states);    # Close the database
format STDOUT_TOP=
Abbreviation     State
==============================
.
format STDOUT=
@<<<<<<<<<<<<<<@<<<<<<<<<<<<<<<
$key,          $value
.
format SUMMARY=
==============================
Number of states:@###
                $total
.
$~=SUMMARY;
write;

----------------------------------------------------------------------


(Output)
  Abbreviation    State
==============================
  AR              Arizona
  CA              California
  ME              Maine
  NE              Nebraska
  TX              Texas
  WA              Washington
==============================
      Number of states:   6
```

解释

1. AnyDBM_File 模块会根据具体的安装版本来选择合适的 DBM 库。
2. dbmopen 函数负责把一个 DBM 文件与一个散列绑定到一起。此处数据库文件名为 statedb，而该散列则名叫 %states。
3. 在打开 DBM 文件后，用户可以访问其键 / 值对。这里把 sort 函数与 key 函数结合起来使用，以便对散列 %states 中的键进行排序。
4. foreach 循环迭代遍历存储键的列表。
5. 每当遍历经过循环时，就从散列 %states 中检索另外的值，这些值是与 DBM 文件连接在一起的。
6. 在给变量 $total 添加一个值后（跟踪 DBM 文件中的项目数），调用 write 函数报告温度情况，并生成格式化后的输出。
7. 关闭 DBM 文件，即撤销散列 %states 与 DBM 文件之间的关联。
8. 这是一个格式模板，用于在每页的顶部放置一个标题。
9. 通过一个点号结束模板的定义。
10. 这是为所有打印到标准输出的页面主体而准备的格式模板。下面的图像行负责格式化位于图像下面各行的键 / 值对。
11. 在报表的底部调用格式模板。

15.2.3 从 DBM 文件删除项

若要清空整个 DBM 文件，用户可使用 undef 函数；例如，undef %states 会清除示例 15.8 所创建的 DBM 文件中所有的项。如果只需删除其中一个键 / 值对，则可使用 delete 函数，并为它提供连接到 DBM 的散列中合适的键。

示例 15.8

```
(The  Script)
    #!/bin/perl
    # dbmopen is an older method of opening a dbm file but simpler
    # than using tie and the SDBM_File module provided
    # in the standard Perl library Program name: remstates.pl
1   use AnyDBM_File;
2   dbmopen(%states, "statedb", 0666) || die;
    TRY: {
        print "Enter the abbreviation for the state to remove. ";
        chomp($abbrev=<STDIN>);
        $abbrev = uc $abbrev;  # Make sure abbreviation is uppercase
3       delete $states{"$abbrev"};
        print "$abbrev removed.\n";
        print "Another entry? ";
        $answer = <STDIN>;
        redo TRY  if $answer =~ /Y|y/;  }
4   dbmclose(%states);

(Output)
5   $ remstates.pl
    Enter the abbreviation for the state to remove. TX
    TX removed.
    Another entry? n
```

```
6   $ getstates.pl
    Abbreviation     State
    ==============================
    AR              Arizona
    CA              California
    ME              Maine
    NE              Nebraska
    WA              Washington
    ==============================
    Number of states:   5

7   $ ls
    getstates.pl    makestates.pl   rmstates.pl     statedb.dir
statedb.pag
```

解释

1. AnyDBM_File 模块会根据具体的安装版本来选择合适的 DBM 库。
2. dbmopen 函数负责把一个 DBM 文件与一个散列绑定到一起。此处数据库文件名为 statedb，而该散列则名叫 %states。
3. 在打开 DBM 文件后，用户可以访问其键 / 值对。delete 函数能够删除与 DBM 相连接的散列 %states 中对应于指定键的相应值。
4. 关闭 DBM 文件，即撤销散列 %states 与 DBM 文件之间的关联。
5. 执行 Perl 脚本 remstates.pl，从 DBM 文件中删除 Texas。
6. 执行 Perl 脚本 getstates.pl，显示 DBM 文件中的数据，可以看到 Texas 已被删除。
7. 清单列出了为产生这些示例结果而需要创建的文件。其中最后两个是由 dbmopen 创建的 DBM 文件。

示例 15.9

```
1   use Fcntl;
2   use SDBM_File;
3   tie(%address, 'SDBM_File', 'email.dbm', O_RDWR|O_CREAT, 0644)
    || die $!;
4   print "The package the hash is tied to: ",ref tied %address,"\n";

5   print "Enter the email address.\n";
    chomp($email=<STDIN>);
6   print "Enter the first name of the addressee.\n";
    chomp($firstname=<STDIN>);
    $firstname = lc $firstname;
    $firstname = ucfirst $firstname;
7   $address{"$email"}=$firstname;
8   while( ($email, $firstname)=each(%address)){
        print  "$email, $firstname\n";
    }
9   untie %address;
```

解释

1. 使用文件控制模块对 DBM 文件执行必要的操作。这里定义了 O_RDWR 和 O_CREAT 标志，在创建 DBM 文件时将要用到这些标志。
2. 这里用到了 SDBM_File 模块。这是由标准 Perl 提供的 DBM 实现，能跨平台运行在多种系统平台上。

3. 本示例没有使用 dbmopen 函数创建或访问 DBM 文件，而是使用了 tie 函数。散列 %address 连接到了包 SDBM_File 上。其 DBM 文件名是 email.dbm。如果该数据库不存在的话，O_CREATE 标志会让程序创建它，并赋予它可读可写（O_RDWR）的权限。
4. 如果调用成功，tie 函数会返回真。ref 函数将返回散列连接到的包名。
5. 在本示例中，使用电子邮件地址作为散列 %address 的键，而相应的散列值则是用户姓名。由于键始终是惟一的，因此这种机制可以避免存入重复的电子邮件地址。要求用户提供输入。
6. 要求用户对各个键提供相应的值。
7. 在这里为数据库赋予新的项。向它赋予键 / 值对，并保存到 DBM 文件中。
8. each 函数将从连接到 DBM 文件的散列中取出键 / 值对。显示 DBM 文件内容。
9. untie 函数负责撤销散列和 DBM 文件之间的关联。

15.3 读者应当学到的知识

1. 读者应当了解 Perl 是如何使用 tie 机制来把标量、数组和散列转换为对象的。
2. tie 函数会返回什么内容？
3. 为什么说 DBM 很有用？
4. DBM 文件一般保存在哪里？

15.4 下章简介

尽管 DBM 文件能在用户存储大批记录时节省大量的时间和空间，但用户可能最终还是感到，自己真正需要的应当是一个关系型数据库，如 Oracle、Sybase 或 MySQL。下章内容将介绍关系型数据库（尤其是 MySQL），以及 Perl 是如何与这些数据库实现交互的。读者将会学习如何使用 DBI 模块，还有一些负责在 MySQL 数据库中连接、查询、刷新和删除记录的方法。

第 16 章　CGI 和 Perl：超级活力双雄

16.1　静态和动态 Web 页面

读者在浏览 Internet 时，会从一个站点跳到另一个站点，浏览各种各样的 Web 页面。这些页面可能是简单的主页，也可能是像 Google 或 Amazon 这样完善的网站。哪怕最简单的 Web 页面在本质上都不过是一个文件，它含有 HTML、文本信息、格式化指令以及名叫链接（links）的带有下划线的短语。链接允许用户连接到其他文档上，这些文档可以位于同一台计算机中，也可能位于网络中的其他计算机上。文档（又称作超文本文档）负责告诉浏览器如何显示文档内容；譬如，使用哪种字体、颜色与样式。在页面中还可能含有超媒体（hypermedia）信息，包括图像、声音、影视乃至指向其他文档的热链接（hotlink）。Web 页面可在文本编辑器中创建，这样得到的 HTML 文件又称作源文件（source file）。用户只需在浏览器的菜单栏中点击“查看”，便可查看这些源文件。

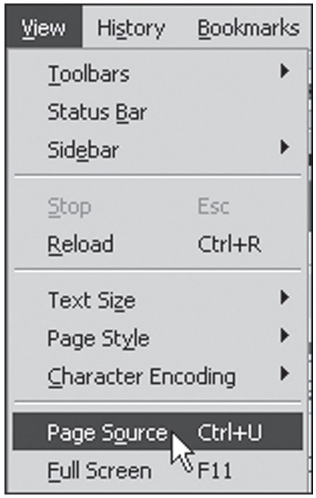


图 16-1　查看源文件

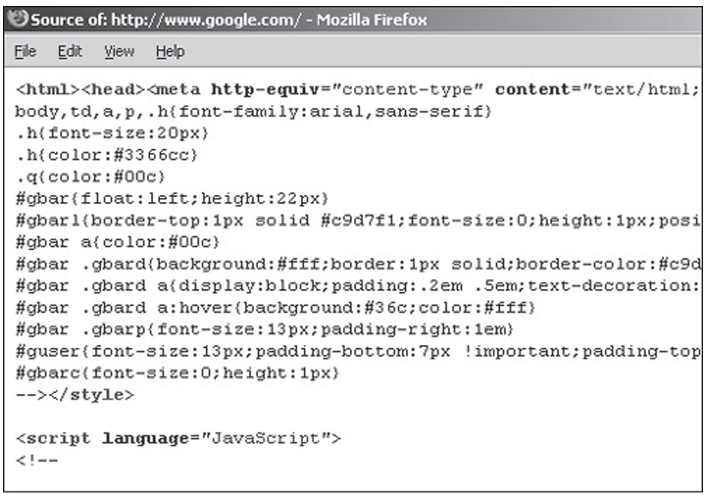


图 16-2　查看 google.com 的部分源文件

为了让浏览器能正确识别页面文件，其文件名应当以 .html 或 .htm 结尾。HTML 标签负责告诉浏览器如何在屏幕上显示文档内容。学会基本的 HTML 语法并不困难，但是读者若想设计开发出精彩有趣的站点，那可就另当别论了。目前世界上有数以千计的公司专注于为其他企业开发 Web 站点，并力图在这片竞争激烈的市场站稳脚跟。

有两种不同类型的页面：静态页面和动态页面。静态页面并不要求与用户实现交互，其作用主要是向用户发送业已存在的文档。它们类似于书本中的页面，一般都负责描述某个公司提供的服务。它们可能会制作得非常精美且非常有趣，但无法按照用户需求去处理信息。另一方面，动态页面则是“活”的，它们能够接受和检索来自用户的信息、生成特殊定制的内容、搜索文本、查询数据库、乃至动态地生成文档。它们还可以根据不同用户的请求不断改变生成的信息。这些动态页面不仅包含 HTML 文本文件，还受程序、脚本的驱动。这些程序或脚本负责与服务器实现交互，并将各类信息发送到用户的浏览器中。为了在程序和服务器之间往返发送数据，需要使用一个服务端程序。服务器本身会把用户请求转发给服务端程序，并由它负责管理这些信息，譬如从表单解析输入数据，或者从某个文件或数据库中检索出数据作为用户请求的结果，最后再将这些数据发送回服务器。

CGI（公共网关接口）协议定义了服务器应当如何与这些程序进行通讯。其功能是允许 Web 服务器越过它的常规边界，以便从外部数据库或文件中检索和访问信息。它还定义一套规范，明确了如何把数据从脚本传送到服务器，或者从服务器传给脚本。网关程序（gateway program）又称作 CGI 脚本，它可以用任何语言编写，但是 Perl 已经成为它事实上的标准语言。这主要应当归功于 Perl 出色的灵活性和易用性。

如果读者已经阅读了前面的章节，就应当很容易地获得 Perl 解释器，也明白它是可移植的。此外，读者一定也了解了 Perl 处理正则表达式、文件、套接字和 I/O 的功能。在了解了 Perl 之后，编写 CGI 脚本就是小菜一碟了。这里关键是要确保正确地安装了服务器和 Perl，并将脚本放到服务器能够找到的目录下。此外还必须正确地设定好路径名，以便让服务器知道脚本保存在何处，以及如何拿到所需要的库。如果安装和实现的步骤不正确，那么其后所有的努力就都是白费力气。如果读者发现浏览器正在抱怨没有找到请求的文件、程序执行遭到禁止、或者文档里面空空如也的话，那可真是一件令人灰心的事情。

编写本章的目标并不是要让读者成为精通 Web 的设计人员，而是希望读者能大概了解 Perl 是如何适应 CGI 模块以及如何创建动态 Web 页面的。有的时候总体设计图是理解设计目标和计划的关键所在。Internet 上充斥着海量的 Web 相关信息，其他很多书籍也对这方面内容有更为深入的介绍。

今天，服务器本身就内置了 PHP 和 ASP.net 等服务器编程语言，使得对 CGI 脚本的需求越来

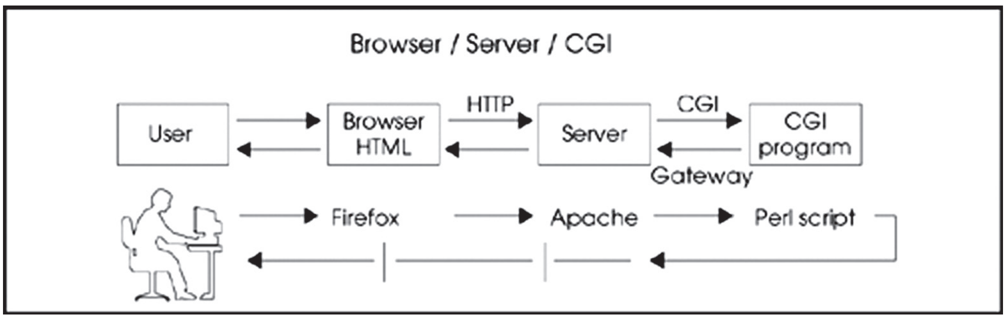


图 16-3　浏览器、服务器和 CGI 程序之间的关系

越淡化。不过鉴于 Perl 的流行程度，再加上现在还有大量的 CGI 脚本尚在使用中，因此就出现了 mod_Perl（外号“Perl 兴奋剂”）。该组件允许用户把 Perl 解释器嵌入到 Apache 服务器中，从而提升 Perl 的性能，并让那些喜欢用 Perl 创建动态页面的开发者能够享受到更快更强大的 CGI 脚本功能。如需了解有关 mod_Perl 的更多信息，请读者参阅附录 D 和 http://Perl.apache.org/。

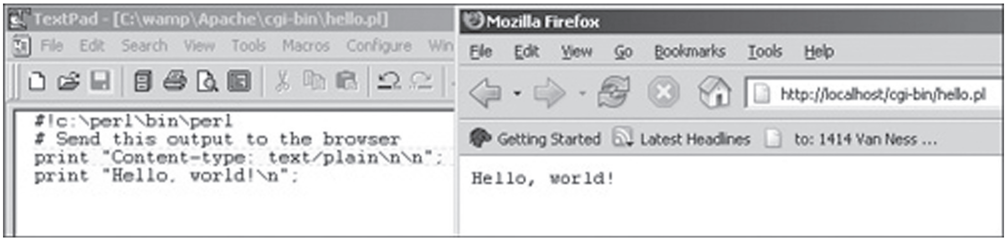


图 16-4　一段向浏览器发送输出的简单 CGI 程序

16.2　工作原理

客户端与服务端之间的 Internet 通信

HTTP 服务器。我们将在第 20 章“网上发送”讨论用于常规网络操作的客户 / 服务器模型和 TCP/IP 协议。Internet 上的通讯都是由 TCP/IP 连接负责处理的。整个 Web 都以此为基础。服务器端会响应客户端（浏览器）的请求，并通过发送文档、执行 CGI 程序或抛出出错消息的方法提供反馈。Web 使用超文本传输协议（HyperText Transport Protocol）即 HTTP 协议来确保服务器端和客户端互相能理解对方在说什么。不过这并不影响 TCP/IP 协议的实现。HTTP 对象会映射到传输的数据单元中，该过程超出了本书的讨论范围。这是一个简单而直观的过程，通常 Web 用户都不会觉察它（有关 HTTP 协议的技术细节请参阅 www.cis.ohio-state.edu/cgi-bin/rfc/rfc2068.html。）HTTP 协议是为了方便 Web 处理超媒体信息而建立的，它是一种面向对象和无状态的协议。在面向对象技术中，文档和文件都称作对象，而与 HTTP 协议相关的操作都称为方法。当协议是无状态时，客户端和服务器端都不保存对方的信息，而只是管理自己的状态信息。

一旦在 Web 服务器和客户端之间建立了 TCP/IP 连接，客户端便可向服务器请求某种服务。Web 服务器一般都运行在众所周知的 80 端口，通过在请求中发送 Accept 语句，客户端可以告诉服务器可处理何种数据。例如，一个客户端可能只能接受 HTML 文本，而另一个客户端可以接受声音、图像以及文本。服务器会尝试处理客户端发送的请求（请求和回应内容都是 ASCII 文本），并将信息发回给客户端（浏览器）。

示例 16.1

```
(Client's (Browser) Request)
GET /pub HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.0 Gold
Host: severname.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,*/*
```

示例 16.2

```
(Server's Response)
HTTP/1.1 200 OK
Server: Apache/1.2b8
Date: Mon, 22 Jan 2007  13:43:22 GMT
Last-modified: Mon, 01 Dec 2007 12:15:33
Content-length: 288
Accept-Ranges: bytes
Connection: close
Content-type: text/html

<HTML><HEAD><TITLE>Hello World!</TITLE>
           ---continue with body---
</HTML>
Connection closed by foreign host.
```

响应会确认所用的 HTTP 版本、用于描述服务器尝试结果（成功或失败）的状态代码、首部和数据。消息的首部表明该请求是否正确，返回什么样的数据类型（例如，其内容类型可能是 html/text），以及要发送多少个字节。消息的数据部分含有实际要发送的文本内容。

然后，用户便可在屏幕上看到格式化的页面。在页面中含有指向其他页面的高亮超链接。不管用户是否点击这些超链接，只要文档显示完毕，这一次事务就算完成了，TCP/IP 连接会马上关闭。在关闭连接后，如果出现了其他请求，则会另外创建新的连接。客户端和服务器并不关心前一次事务的内容，也就是说协议是无状态的。

HTTP 协议还可用于在浏览器、代理和网关之间进行通信，其中的网关可以连接到由 FTP、Gopher、WAIS 和 NNTP 等协议支持的其他 Internet 系统上。

HTTP 状态代码和访问 Log 文件。当服务器响应客户端时，会发送含有请求处理方式的信息。如果其状态代码取值在 100 到 300 之间，大多数 Web 浏览器都会默默地处理它们。小于 100 的状态代码表明正在处理服务器请求。最为常见的状态代码是 200，表明请求处理成功，也就是说成功地接受并满足了请求信息。

读者不妨查看一下服务器上的访问日志，便可观察到服务器在完成一个事务后会发送什么样的代码[⊖]（HTTP 状态代码列表详见表 16-1）。下面的示例取自 Apache 服务器访问日志的摘录。该日志记录了请求信息以及作为请求结果而生成的状态代码。出错日志中含有一般会由程序发送到屏幕上的标准出错消息，譬如语法或编译错误。

表 16-1　HTTP 状态代码

状态	代码消息
100	继续（Continue）
200	成功（Success，OK）
204	无内容（No Content）
301	文档被移动（Document Moved）
304	文档未被移动，无消息体（Document Not Modified，No Message Body）

⊖ 有关状态代码的详细知识，参见 www.w3.org/Protocol/HTTP/HTRESP.html

（续）

状态	代码消息
400	错误请求（Bad Request）
401	非法授权（Unauthorized）
403	禁止（Forbidden）
404	无法找到（Not Found）
405	不允许的方法（Method Not Allowed）
500	内部服务器出错（Internal Server Error）
501	未曾实现（Not Implemented）
503	服务不可用（Service Unavailable）

示例 16.3

```
(From Apache's Access log)

1   127.0.0.1 - - [22/May/2007:20:50:42 -0700] "GET /cgi-bin/firstc
gi.pl HTTP/1.1" 200 235

2 127.0.0.1 - - [22/May/2007:20:50:43 -0700] "GET /Williewonker.j
pg HTTP/1.1" 304 -

3 127.0.0.1 - - [22/May/2007:20:50:52 -0700] "GET /cgi-bin/env.pl
x HTTP/1.1" 500 623
```

解释

1. 服务器主机名为 127.0.0.1，其后是两个短横表示未知值，如用户 ID 和密码。这里记录了很多信息，包括其请求的时刻，其请求类型为 GET（详见“GET”一节），其访问的文件是 firstcgi.pl。其协议为 HTTP/1.1。其服务器发出的状态代码是 200，表示处理成功！该请求已经处理完毕。
2. 状态代码 304 表明该请求将得到一个不含消息体且不能修改的文档。在本例中，该文件就是一个 jpeg 图像文件。
3. 状态代码 500 表明发生了内部服务器错误，也就是说服务器内部的某个地方出现了错误，譬如 Perl 程序中的语法错误，或者在 #! 行中指定了不正确的全路径。浏览器的请求没有得到满足。发送的字节一共是 623 个。

URL（统一资源定位符）。URL 用于浏览 Web。用户只需单击热链接，便能访问新的页面;或者也可在浏览器的地址栏中输入 URL，就能打开文件或运行脚本。它是一种虚拟地址，负责指定页面、对象、脚本等实体的位置。它引用了现有的协议，譬如 HTTP、Gopher、FTP、mailto、file、Telnet 或 news（详见表 16-2）。典型的基于当前流行的 Web HTTP 协议的 URL 格式如下：

http://www.comp.com/dir/text.html

表 16-2　Web 协议

协议	功　能	示　例
http:	超文本传输协议（HyperText Transfer Protocol）	http://www.nnic.noaa.gov/cgi-bin/netcast.cgi 打开 Web 页面或启动 CGI 脚本
ftp:	文本传输协议（File Transfer Protocol）	ftp://jague.gsfc.nasa.gov/pub
mailto:	基于 e-mail 地址的邮件协议	mailto:debbiej@aol.com
file:	打开本地文件	file://opt/apache/htdocs/file.html
telnet:	打开 Telnet 会话	telnet://nickym@netcom.com
news:	通过新闻服务器打开新闻会话	news:alt.fan.john-lennon Name or Address

URL 中提供的两项基本信息是协议 http 和该协议需要的数据，如 http://www.comp.com/dir/text.html。表 16-3 进一步定义了 URL 的各组成部分。

表 16-3　URL 各部分

组成部分	描　述
协议	诸如 HTTP、Gopher、FTP、Telnet 和 news 之类的服务
主机 /IP 号	DNS 主机名或 IP 地址
端口	服务器使用的 TCP 端口号，一般都是 80 端口
路径	服务器上对象的路径和文件名引用
参数	服务器上对象使用的具体参数
查询	CGI 脚本的查询串
段	指向对象子集的引用

默认的 HTTP 网络端口是 80；如果 HTTP 服务器驻留在其他网络端口上，如 www.comp.com 上的 12345 端口，则其 URL 应当写作：

http://www.comp.com:12345/dir/text.html

并不是每个 URL 部分都是必需的。如果读者是在 Netscape 浏览器的定位器（Locator）栏中搜索文档，则对 URL 无需提供端口号、参数、查询或段。如果 URL 是 HTML 文档中的一个热链接，则可能含有指向下一个文档的相对路径，即相对于服务器根目录的路径。如果用户需要填写表单（form），则 URL 中就可能出现带有问号的信息。URL 的外观实际上取决于它所使用的协议，以及需要完成的操作。

示例 16.4

```
1   http://www.cis.ohio-state.edu/htbin/rfc2068.html
2   http://127.0.0.1/Sample.html
3   ftp://ptgp023@ptgpftp.pearsoned.com/quigley
4   file:///c:/wamp/www/family.jpg
5   http://localhost/cgi-bin/form.cgi?name=Fred+Thompson
```

解释

1. 协议是 http。

主机名 www.cis.ohio-state.edu/htbin/rfc2068.html 由如下部分组成[a]：

由域名服务器（DNS）翻译为 IP 地址的主机名；
域名是 ohio-state.edu
其顶级域名是 edu；
保存 HTML 文件的目录是 htbin；
要检索的文件是 rfc2068.html，它是一个 HTML 文档。
2. 协议是 http。
这里使用 IP 地址而不是主机名；这是本地主机的 IP 地址；
文件位于服务器上的文档根目录中。文件由 HTML 文本构成。
3. 协议是 ftp。
FTP 服务器是 ptgpftp.pearsoned。
顶级域名是 com。
目录为 quigley。
4. 协议是 file，打开本地文件。
主机名缺失，引用本地主机名。
列出指向 index.html 文件的全路径。
5. 问号后面的信息是 URL 的查询部分，可从表单提交的输入内容中获得。其查询串内容是一段 URL 编码。在本例中，位于 hello 与 there 之间的空格将由加号替代。服务器会将该查询部分保存到环境变量 QUERY_STRING 中。然后可以将它传递给 HTML 文档所调用的 CGI 程序（详见“GET 方法”一节）。

a. 大多数 Web 服务器的主机名都以 www 开头，但这只不过是一个惯例而已。

文件 URL 和服务器根目录。如果 URL 中用到的协议是 file，服务器就假定该文件位于本地主机上。URL 中含有文件名及其全路径。如果在协议后面出现了服务器名，则所有路径都将与服务器上的文档根目录相关。文档根目录定义在 Web 服务器的主目录下面，持有服务器的配置文件、错误信息以及日志文件。用户可在文档根目录中储存 HTML 文档、图像、以及服务器需要的其他任何文档；譬如，名叫“htdocs”或“www”的文件。

与 UNIX 中以根目录为起点的绝对路径不同，这里位于路径前面的正斜杠并不是路径的组成部分，而是负责将路径与主机名分开的分隔符。一个用于表示服务器根目录中某个文档的 URL 示例如下所示：

http://localhost/index.html

其全路径名可以是：

C:/wamp/www/index.html

所谓部分 URL 或相对 URL，指得是链接到同一个服务器上的文档的快捷方式。例如，如果一个位于 http://www.myserver/stories/webjoke.html 的文档含有一个指向 images/webjoke.gif 文件的链接，那么这就是一个相对 URL。浏览器会把这个相对 URL 扩展为绝对 URL，即 http:// www.myserver/stories/images/webjoke.gif，并在需要时请求该文档。

16.3 使用 HTML 创建 Web 页面

为了编写 Web 页面，用户必须首先学习一下 HTML 语言。有关这个主题已经有了大量的著作。

这里我们只是简单介绍一下 HTML 的基础知识，并介绍如何编写带有表单和 CGI 脚本的简单页面。

正如前面所介绍的，在 HTML 中编写的 Web 页面是一些 ASCII 文本文件。HTML 由称作标签（Tag）的指令所构成，能够告诉浏览器如何去显示页面中的文本[⊖]。当用户输入 URL 或者点击页面上的热链接时，浏览器（客户端）就会通知服务器它需要哪个文件，并由服务器负责把该文件返回给浏览器。含有 HTML 内容的文本由纯文本、图像、影像、声音和热链接构成。然后由浏览器负责解释 HTML 标签，并在屏幕上显示格式化后的页面（如要查看 Web 页面的源文件，在 Netscape 浏览器中可用 View 菜单里的 View Document 功能；而在 Internet Explorer 中则可选择 View 菜单，然后选择 Source 项，这样便能查看产生页面的 HTML 标签）。

创建标签。用户可以使用任何一种文本编辑器来创建 HTML 源文件。如果一个文件以 .html 或 .htm 结尾，就说明它是一个 HTML 文件。HTML 标签都位于角括号 <> 之间，负责描述文档的显示方法。标签是很容易理解的。例如，如果要创建一个标题，用户应当把标签放在角括号内，并在起始标签 <TITLE> 和结束标签 </TITLE> 之间提供实际的标题内容文本。下面这行内容又称为 TITLE 元素，它由开始标签 <TITLE>、包含的文本和结束标签 </TITLE> 组成。标签中还可带有属性，用于进一步描述其功能。例如，文本输入框标签允许设定其行数和列数，而图片标签则可以在页面顶部居中对齐。元素和属性都是大小写敏感的。

<TITLE> War and Peace </TITLE>

当浏览器看到这个指令时，就会在浏览窗口的顶部一栏中打印出标题内容，作为本页面的标题。如果需要在 HTML 中置入注释，则可把注释内容插入到 <!-- 和 --> 之间。

由于 HTML 是一种结构化的语言，因此在文档中放置标签时，必须遵守一些规则。本书将在下面讨论这些规则。

简单 HTML 文档。 读者可以在自己最喜欢的文本编辑器中创建下面这个 HTML 文件。该文件由一组简单的标记元素（tagged element）构成。浏览器将在显示该文件时负责渲染这些元素。详见 http://www.w3.org/MarkUp/Guide/。表 16.4 列出了所有的简单 HTML 标签及其功能。

示例 16.5

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Simple HTML Template</title>
</head>
<body bgcolor="silver">
    <div align="center">
    <h1>Getting Started</h1>
    <p>This is a paragraph in the body of my document.
        It starts and ends with a paragraph tag.
        </br>You can put pictures, links, movies, here,
        whatever you like..
    </p>
    <p>
        <img src="file:///c:/wamp/www/mexico.jpg" border="1"
            width="200" height="250">
    </p>
```

[⊖] 如果读者曾经用过 UNIX 中的 nroff 或 troff 程序来格式化文本的话，就会很快认出用于格式化 HTML 的这些标签。

```
        <br/>
        For a great HTML tutorial:
12          <a href="http://www.w3schools.com/html/default.asp">
            <strong>Click here</strong></a>
        </div>
13  </body>
14  </html>
```

解释

1. 这一行声明了 DOCTYPE；即该文档使用的是那一种 HTML 或 XHTML。在后面针对某个特定版本验证语法时，将会用到这个声明信息。对于纯 HTML4，可以把 DOCTYPE 声明为 transitional。其他可选的值包括 frameset 和 strict，但它们都超出了本书介绍的范围。
2. HTML 文档的所有文本内容都位于起始标签 <html> 和结束标签 </html> 之间。尽管 HTML 是创建 Web 页面时的标准语言，但还是存在着其他类似于 HTML 的页面创建语言。HTML 元素表明了这是一个 HTML 文档。即便读者抛弃这个标签，浏览器也不会有什么意见，只不过加上它显得更正式一些。
3. 在 <head> 和 </head> 之间插入一些文档信息，譬如文档的标题。这些信息将不和其他内容一道显示出来。<head> 标签一般都紧紧跟随在 <html> 标签后面。
4. <title> 标签用于创建文档标题，该标题将显示在浏览器窗口的顶部。
5. 这是对应于 <head> 标签的结束标签。
6. 文档主题位于起始标签 <body> 和结束标签 </body> 之间，它将显示在浏览器窗口中。
7. <div> 标签负责在页面中创建一块分隔区域。所谓位于起始标签 <div> 和结束标签 </div> 之间的内容都会居中显示。
8. 所有的一级标题都位于起始标签 <h1> 和结束标签 </h1> 之间。
9. <p> 标签表明新起一个段落。</p> 表明段落到此结束。
10. 本行使用了
 标签在文本中实现换行。
11. <img> 标签允许用户在文档中置入 logo 标志、照片、插图、动画等，它们可以是 gif、png

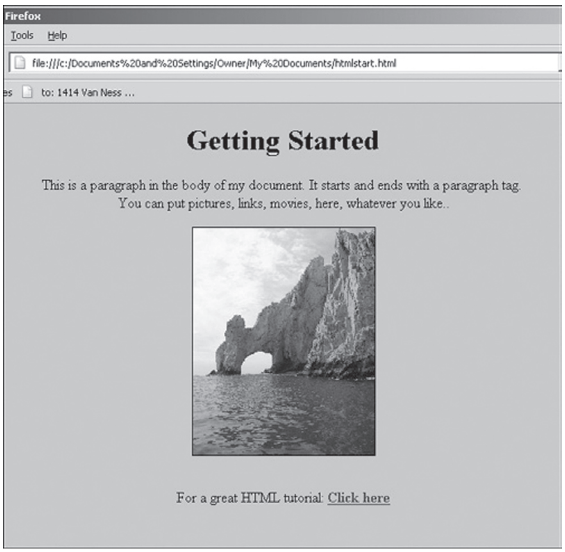


图 16-5　示例 16-5 中的 HTML 文档在 Mozilla Firefox 中的显示效果

或 jpeg 图像文件。tag 标签附带的 src 属性指明了文档在服务器上的位置。通过指定高度和宽度象素数，读者也可自由缩放图像到所需的尺寸。

12. 读者还可通过 <a href> 标签在页面上创建链接。当用户点击该链接时，就会自动重定向到一个有关如何使用 HTML 的页面上。
13. 这个标签表明文档的主体部分在此结束。
14. 这个标签表明 HTML 文档到此结束。

表 16-4 简单 HTML 标签及其功能

标签元素	功 能
<!--text-->	注释文本，不显示
<*BASE HREF*="http://www.bus.com/my.html">	存储文档的位置
<*HTML*>document</*HTML*>	位于文档开头和末尾，向浏览器表明这是一个 HTML 文档
<*HEAD*>heading info</*HEAD*>	文档内的第一个元素，含有标题、元标签（metatag）、JavaScript 以及 CSS。只有标题能直接显示
<*TITLE*>titleof the document</*TITLE*>	文档的标题；显示在屏幕顶部或窗口中文档文本的外面，可置于书签列表中
<*BODY*>document content</*BODY*>	含有显示的所有文本和其他对象
<*H1*> heading type </*H1*>	创建一到六级标题的粗体元素，其级别分别是 H1、H2、H3、H4、H5、H6，其中 H1 是顶级标题
<*P*>text</*P*>	段标记，表明段的开始；文本块之后插入分隔符。它可以位于一行中的任意位置。其结束标签是可选的。在碰到 </P> 或另一个 <P> 标签（标记新段）时，本段才算结束
<*B*>text</*B*>	粗体文本
<*I*>text</*I*>	斜体文本
<*TT*>text</*TT*>	打字机型文本
<*U*>text</*U*>	下划线文本
<*BR*>	行分隔符
<*HR*>	水平阴影行
<*UL*>	无序列表的起始标签
<*LI*>	列表中的一项
<*LI*>	列表中的另一项
</*UL*>	无序列表的结束标签
<*OL*>	有序列表的起始标签
<*DL*>	描述性列表
<*DT*>	描述性列表中的一项
<*DT*>	描述性列表中的另一项
</*DL*>	描述性列表的结束标签
<*STRONG*>	粗体文本
<*EM*>	斜体文本
<*BLOCKQUOTE*>text</*BLOCKQUOTE*>	引用前后使用空格，使大块文本成为斜体
<*A HREF SRC="URL"*>	创建热链接，指向 Web 上位于 URL 的资源
<*IMG SRC="URL"*>	将图像加载到 Web 页面中，URL 是图像文件的地址

16.4 HTML 和 CGI 如何协作

前面我们已经讨论过，HTML 是用于确定 Web 页面显示方式的标记语言。CGI 则是扩展服务器功能的协议。服务器上执行的 CGI 程序主要负责处理表单，如注册表单或购物单。如果读者曾经在 Amazon.com 购买过书籍或 CD 唱片，就应当知道什么是表单。当浏览器（客户端）发出请求时，服务器会检查 URL。如果服务器在路径中看到了 cgi-bin 目录，便会进入该目录，然后打开一个管道，并执行 CGI 程序。CGI 程序将从管道中获得输入，并通过管道将标准输出返回给服务器。其标准错误信息则会发送到服务器的错误日志中。如果 CGI 需要与服务器进行对话，则必须使用 Web 语言，因为这是在浏览器上最终用于显示 Web 页面的语言。然后，CGI 程序会使用 HTML 标签对数据进行格式化，并将它们发送回 HTTP 服务器。最后由服务器把文档返回给浏览器，渲染其 HTML 标签，并显示给用户。

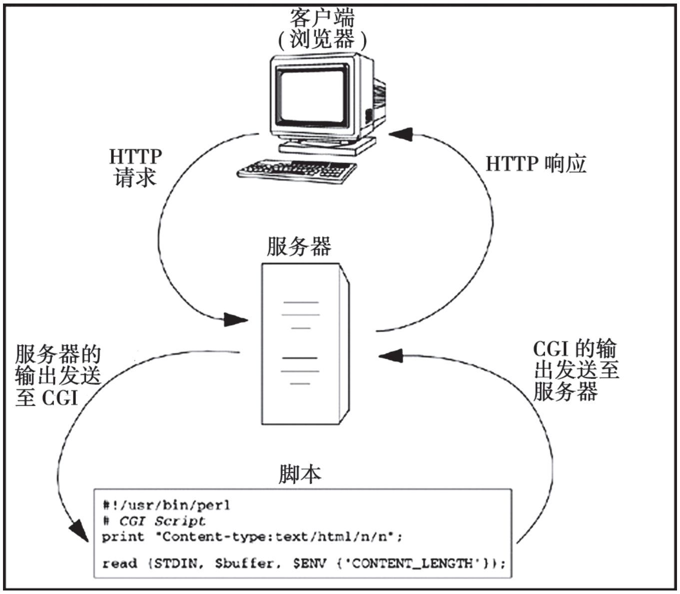


图 16-6　客户端 / 服务器 /CGI 程序之间的关系

16.4.1 简单 CGI 脚本

在编写更为复杂的动态页面之前，我们首先看一个简单的 CGI 脚本，该脚本由 Perl 编写，执行在一个 Apache 服务器上。当从浏览器访问它时，该脚本会随时刷新当前日期和时间。该脚本由一系列 print 语句组成，其中大部分语句都将 HTML 输出到 stdout（通往服务器的管道），正如图 16-6 所示。其中有些特性使得这段脚本不同于通常的 Perl 脚本：

该程序是从 CGI 目录 cgi-bin 处直接执行的。

该脚本的第一行又称作“shbang”行，负责告诉服务器 Perl 解释器位于哪里。该行如果出错，就会造成“Internal Server Error”错误。

CGI 脚本中的另一行重要内容则是其标题行，又称作 HTTP MIME 首部。它将告诉浏览器本示例的程序会发送哪种类型的输出内容；譬如，如果把“Content-type: text/html”发送给浏览器，就

说明该文档中即含有常规文本又含有 HTML 标签。本章后面将详细介绍相关细节。

首部行（header line）后面必须出现一个空行。为了达成该效果，读者可以在行末添加“\n\n”。如果是在 here 文档中，则也可在文本编辑器中模仿前面所述的简单 Perl CGI 脚本提供一个空行。

示例 16.6

```
1 #!c:/ActivePerl/bin/perl.exe
2 $now=localtime;
  $myname="Willie";
3  print <<EOF;
4  Content-type:text/html
5   <body bgcolor="silver">
     <div align="center">
     <font face="arial" size="+2" color="darkblue">
     <b>Welcome,  Wee $myname!<p>
     <img src="/Williewonker.jpg" width="200"
          height="230" border="1">
     <p>
6    Today is $now<br>
    </body>
7 EOF
```

解释

1. 第一行很重要。如果缺少这一行，很多 Web 服务器就无法运行 CGI 脚本。第一行负责告诉服务器 Perl 的安装位置。
2. 定义两个标量型变量：$now 负责从 Perl 的 localtime 函数处获得返回值。每当执行该程序时，其时间都会发生变化，从而让该页面成为动态的。变量 $myname 则赋值为“Willie”。
3. 本行启动一个 here 文档（详见第 2 章）。here 文档提供了一种替代引号的机制，允许用户通过一个 print 函数打印多行信息。EOF 是一个用户自定义的终结符，它和第 7 行相互匹配。位于两个 EOF 终结符之间的所有文本都将作为输出内容发送到浏览器，并由后者予以显示。
4. 本行又称作 MIME 首部（MIME header）。不管使用的是哪一种编程语言，CGI 程序的第一行输出都必须是一个 MIME 首部，其后必须跟随两个换行符。本行负责说明程序会发送哪种类型的数据。在这里，CGI 脚本将把 HTML 文本发送回服务器。\n\n 说明打印一个空行。为了成功运行 CGI 程序，这个空行也是非常重要的。所有首部都要求在文档内容前提供一个空行。
5. 这里是混杂着 HTML 标签的文本。浏览器将呈现这些标签，以便提供 HTML 页面。
6. 程序每次执行时，变量值都会不断变化，以便实时显示服务器端的当前时间和日期。
7. EOF 表明 here 文档到此结束。

有的服务器要求 CGI 脚本名必须以 .cgi 或 .pl 结尾，这样才能把它们识别为 CGI 脚本。此外在 UNIX 系统中，当创建好脚本后，还需要开启该脚本文件的执行权限。读者可以在 shell 提示符下键入如下内容：

```
chmod 755 <scriptname>
```

或者

```
chmod +x <scriptname>
```

在浏览器定位窗口中键入的 URL 包括协议、主机名、存有 CGI 脚本的目录、还有 CGI 脚本名。URL 的样式如下所示：

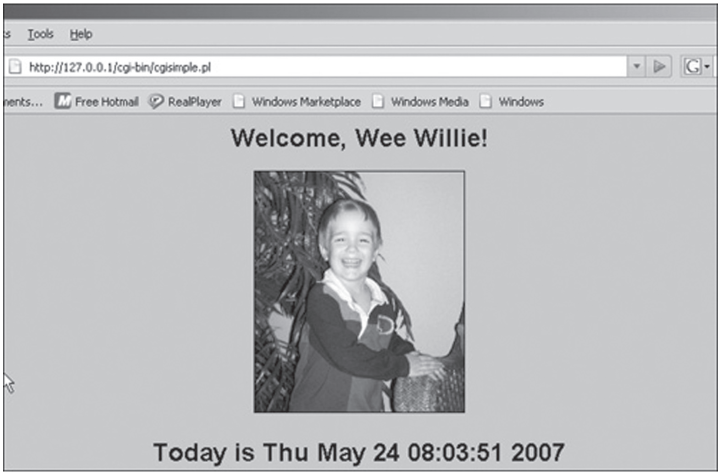


图 16-7　Perl CGI 脚本在浏览器中的输出

```
http://localhost/cgi-bin/cgisimple.pl
```

HTTP 首部。大多 CGI 程序的第一行都是 HTTP 首部，它能告诉浏览器程序会把什么类型的输出内容发送给它。在首部行后，必须有一个空行和两个换行符。首部类型又称作 MIME 类型（MIME 表示多用途 Internet 扩展，即 multipurpose internet extension）。两种最常见的类型是“Content-type: text/html \n\n”和“Content-type: text/plain \n\n”。另一种首部类型叫作位置首部（Location Header），用于将浏览器重定向到其他 Web 页面上。最后，Cookie 首部用于设置 cookie，以便维护浏览器状态；即跟踪维护那些在服务器与浏览器之间的事务关闭后原本应当消失的信息。在首部行后面必须提供一个空白行。在 Perl 中可以通过在行末追加 \n\n 的方式来实现它。表 16-5 列出了常用的 HTTP 首部。

表 16-5　HTTP 首部

首部	类型	值
Content-type:	text/plain	纯文本
Content-type:	text/html	HTML 标签与文本
Content-type:	image/gif	GIF 图像
Location:	http://www....	重定向到其他 Web 页面
Set-cookie:NAMe=VALUE...	Cookie	在客户端浏览器中设置 cookie

16.4.2　错误日志文件

错误日志和 STDERR。一般而言，当 Perl 脚本中出现错误时，应当把错误信息发送到中断屏幕上（STDERR）。但如果是一段由服务器启动的 CGI 脚本，则错误信息不会发送到屏幕，而是发送到服务器的错误日志文件中。在浏览器上，用户将只能看到“Empty Document”或“Internal Server Error”，而这并不能说明程序中到底出了什么问题。

在将脚本传到服务器之前，请读者始终记得在 shell 命令行中使用 -c 开关检查自己脚本的语法。

否则，读者将很难看到错误信息，除非直接查看日志文件。请使用 -c 开关检查 Perl 脚本语法。

示例 16.7

```
(At the Command line)
1   perl -c perlscript
2   perlscript syntax OK
```

示例 16.8

```
(Perl syntax errors shown in the Apache server's error log)

    [Mon Jul 20 10:44:04 2006] access to /opt/apache_1.2b8/
        cgi-bin/submit-form failed for susan, reason: Premature end
        of script headers
    [Mon Sep 14 11:11:32 2006] httpd: caught SIGTERM, shutting down
    [Fri Sep 25 16:13:11 1998] Server configured -- resuming normal
        operations
1   Bare word found where operator expected at welcome.pl line 21,
         near "/font></TABLE"
    (Missing operator before TABLE?)
2   syntax error at welcome.pl line 21, near "<TH><"
    syntax error at welcome.pl line 24, near "else"
    [Fri Sep 25 16:16:18 2006] access to /opt/apache_1.2b8/
        cgi-bin/visit_count.pl failed for susan, reason:
        Premature end of script headers
```

16.5 控制信息出入 CGI 脚本

服务器和 CGI 脚本之间主要通过四种方式进行通信。当浏览器向服务器发出请求后，服务器可以将其转发给 CGI 脚本。CGI 脚本通过如下方式从服务器获得输入：

1. 环境变量
2. 查询字符串
3. 标准输入
4. 附加路径信息

当 CGI 程序从服务器获得输入后，会对它进行分析和处理，然后进行格式化，以便让服务器能将信息反馈给浏览器。CGI 脚本可通过如下方式经网关发送输出内容：

1. 动态生成新文档
2. 将现有的静态文件发送到标准输出
3. 使用 URL，让浏览器重定向到其他位置获取文档

CGI 环境变量

服务器会向 CGI 程序传递许多环境变量。环境变量是在服务器执行网关程序时得以设置的，并且对所有请求都有效。环境变量含有与服务器、CGI 程序、端口、协议和路径等有关的信息。用户输入内容一般都会赋值给 QUERY_STRING 环境变量。在下面的示例中，该环境变量的值为空，因为程序始终没有要求用户提供输入，即 HTML 文档中没有出现 INPUT 标签。

示例 16.9

```
(The CGI Perl Script)

#!/bin/perl
print "Content type: text/plain\n\n";
print "CGI/1.1 test script report:\n\n";
# Print out all the environment variables
while(($key, $value)=each(%ENV)){
    print "$key = $value\n";
}
```

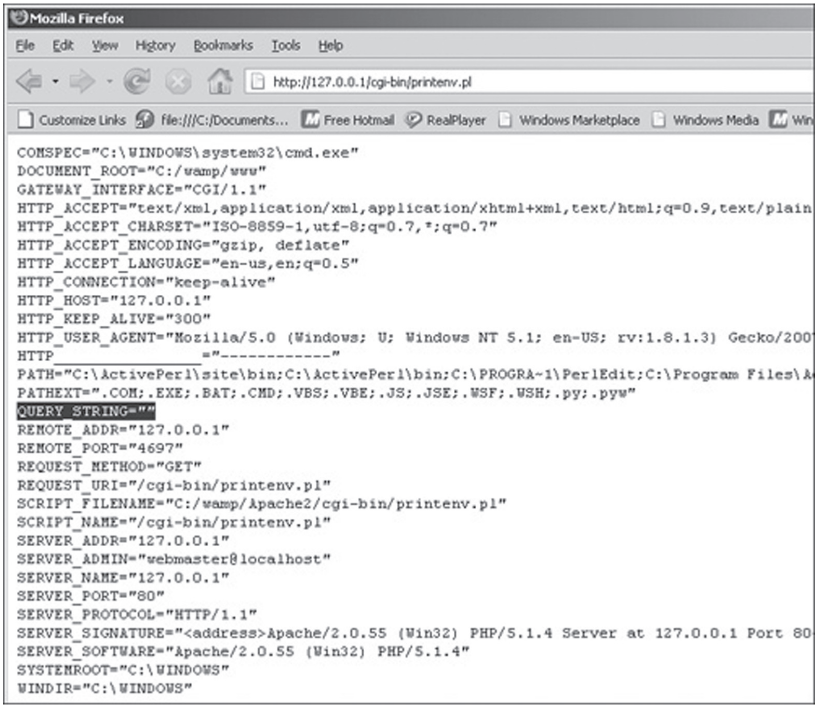


图 16-8　QUERY_STRING 环境变量

不知读者是否注意到，当在浏览器中进行搜索时，其地址行末的内容呈现这样的形式：

http://www.google.com/search?h1=en&q=related%3Awww.lynda.com%2Fhexh.html&btnG=Search

位于？之后的信息都是由浏览器发送给服务器的，它们都将赋值给 QUERY_STRING 环境变量。这是一个经过浏览器编码的 URL；即，字符串中任何的非字母表字符都替换成了 +、& 或者以 % 开头的一个十六进制数字，譬如 %2。服务器上的程序将负责对该字符串进行解码，以使它变得合法。在本书中，该程序是一个 Perl CGI 脚本。当然，它也可以是一个 C、C++、TCL、Shell 脚本等。

在下面的示例中，我们将在示例 16.9 的 CGI 脚本中为 URL 添加一个？及其相应文本字符串。该字符串由浏览器负责编码，并赋值给 QUERY_STRING 环境变量，以便 Perl 脚本能方便地访问它。下面一步则会移除其中的 %20（表示空格的十六进制值）；即，对编码字符串进行解码。所有

这些内容将在后面予以详细说明。不过在这里，读者在看到这种类型的 URL 时，应当明白位于？之后的字符串将会放到 QUERY_STRING 环境变量中，并传递给读者编写的程序。表 16-6 列出了常用的一些 CGI 环境变量。

表 16-6　CGI 环境变量（必须大写）

名称	值	示例
AUTH_TYPE	如果服务器支持用户验证机制，则对用户进行验证	
CONTENT_LENGTH	从服务器传给 CGI 程序的字节数	Content-length=55
CONTENT_TYPE	查询数据的 MIME 类型	text/html
DOCUMENT ROOT	服务器提供 Web 文档的目录	/opt/apache/htdocs/index.html
GATEWAY_INTERFACE	服务器使用的 CGI 修订版本	CGI/1.1
HTTP_ACCEPT	客户端接受的 MIME 类型	image/gif，image/jpeg，等等
HTTP_CONNECTION	首选的 HTTP 连接类型	Keep-Alive

URL 后面追加了一个问号，其后则是一段文本字符串。

带有 CGI 脚本链接的 HTML 文件。　下面的示例是一个 HTML 文件，允许用户通过点击页面上的超链接来打印所有的环境变量。当浏览器显示这些文档时，用户可以点击热链接 here，然后服务器就会执行 CGI 脚本 printenv.pl。在 HTML 文档中，字符串 here 和 URL http://localhost/cgi-bin/printenv.pl 位于标签 <A href></A> 之间。如果用户没有点击该热链接，浏览器便会显示其余文档内容。下面是浏览器显示的 HTML 源文件内容。浏览器的输出如图 16-9 所示。

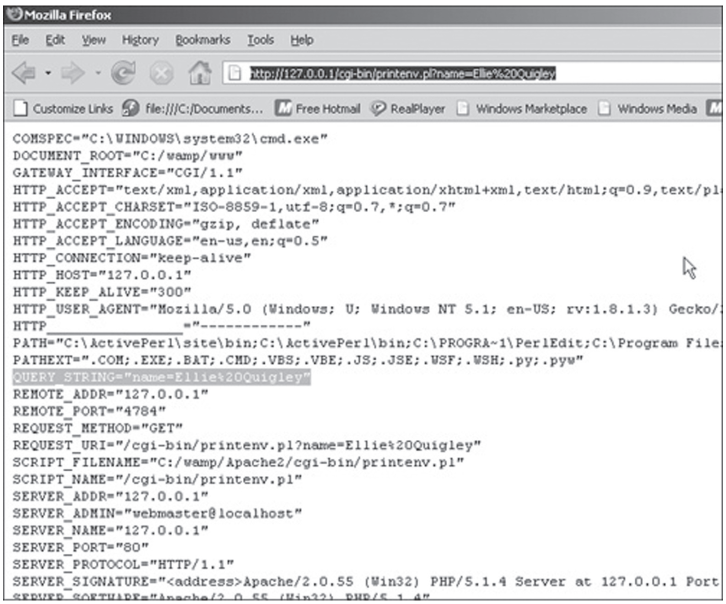


图 16-9　QUERY_STRING 环境变量现在有了值

示例 16.10

```
(The HTML file with a hotlink to a CGI script)
1   <html>
2   <head>
3   <title>TESTING ENV VARIABLES</title>
    </head>
    <body>
4   <p>
    <h1> MAJOR TEST </h1>
    <p> if you would like to see the environment variables<br>
    being passed on by the server, click .
5   <a href="http://localhost/cgi-bin/printenv.pl">here</a>
    <p>text continues here...
    </body>
    </html>
```

解释

1. *<HTML>* 标签表明该文档使用的是 HTML 协议。
2. *<HEAD>* 标签含有标题以及实际文档外显示的信息。
3. 在浏览器窗口的顶栏（top bar）中显示 *<TITLE>* 标签。
4. *<P>* 表明段的开始。*
* 标签负责分隔行。
5. 把 *<A>* 标签赋值为服务器 localhost 上 CGI 脚本 printenv.pl 的路径。浏览器会以带有蓝色下划线的字母显示单词 here。如果用户点击该单词的话，服务器便会执行 CGI 脚本。脚本将打印出由服务器传递给脚本的所有环境变量。这是 Web 服务器向 CGI 脚本传递信息的一种途径。实际的 CGI 脚本如示例 16.9 所示。

16.6 CGI 和表单

处理用户输入是使用 CGI 脚本最常见的原因。这项工作往往通过表单来完成。表单会提供一系列称作虚拟输入设备（virtual input device）的方法，并由它们负责接受用户输入。它们包括单选按钮、复选框、弹出菜单和文本框等。所有表单都位于 HTML 中，以一个 <form> 标签开头，并以 </form> 标签结尾。用户可以指定方法的属性。方法属性表示如何处理表单。其默认方法是 GET，而 POST 方法则提供了最为常用的替代途径⊖。如果不想让操作影响到服务器的状态，譬如执行简单的文档搜索或数据库查询操作，则应首选 GET 方法。如果操作会影响到服务器的状态，譬如添加或删除数据库记录时，则应选用 POST 方法。本书将在下面一节详细介绍这两种方法。当用户点击提交按钮以提交数据时，ACTION 属性便会赋值为指向待执行 CGI 脚本的 URL。

浏览器通过显示可编辑的字段来从用户获取输入信息。这些字段均由 HTML 的 <INPUT TYPE=key/value> 标签创建，其形式可以包括复选框、文本框、单选按钮等。用户输入表单的数据将以名称 / 值（name/value）形式编码为一个字符串，然后发送到服务器。这里的值表达了用户实际输入的值。CGI 程序必须理解这些输入的编码方式，以便能分析和使用它们。我们先来看看如何在浏览器中提供输入，为此需要查看一个简单的文档，以及负责生成该文档的 HTML 代码。这个示例将不处理输入的内容，因此当用户按下 Submit 按钮时，程序会将一个出错消息发送给服务器的

⊖ HTML 标签和表单属性都不是大小写敏感的。

错误日志。浏览器将什么都不显示。其获得输入的默认方法是 GET。

生成表单的所有步骤如下所示：

1. START：以 HTML 的 <form> 标签标识表单的开始。<form> 标签必须嵌套在 HTML 的 <body></body> 标签之间。

2. ACTION：<form> 标签的 action 属性提供了负责处理表单输入数据的 CGI 脚本的 URL。

3. METHOD：负责提供输入数据的方法。默认方法是 GET。

4. CREATE：创建一个带有按钮和框的表单，并可利用 HTML 标签与字段任意美化它。

5. SUBMIT：创建一个 submit 按钮，用于处理整个表单。该按钮将启动 ACTION 属性中列出的 CGI 脚本。

6. END：终止表单和 HTML 文档。

16.6.1 表单输入类型

表 16-7 表单输入类型

输入类型	属性	描述
CHECKBOX	*NAME,VALUE*	显示可以复选的方框。根据用户输入创建相应名称 / 值对。可选多个框
FILE	*NAME*	规定上传到服务器的文件。其 MIME 类型必需是 multipart/form-data
HIDDEN	*NAME,VALUE*	提供名称 / 值对，同时不在屏幕上显示对象
IMAGE	*SRC,VALUE,ALIGN*	与提交按钮相同，但显示图像而非文本。图像是 SRC 指定的文件
PASSWORD	*NAME,VALUE*	与文本框相同，但它是隐藏的。在框中显示星号代替输入的字符
RADIO	*NAME,VALUE*	与复选框类似，但它一次只能选取一个框
RESET	*NAME,VALUE*	将表单重置为初始情况，清除所有输入的字段
SELECT	*NAME,OPTION SIZE,MULTIPLE*	提供弹出菜单和滚动列表。一次只能选择其中之一。属性 MULTIPLE 能创建可见的滚动列表。若 SIZE 为 1，则创建只含单个可见框的弹出菜单
SUBMIT	*NAME,VALUE*	在按下时执行表单；启动 CGI
TEXT	*NAME,SIZE,MAXLENGTH*	创建可供用户输入的文本框。MAXLENGTH 规定其允许的最大字符数
TEXTAREA	*NAME,SIZE,ROWS,COLS*	创建可跨越多行内容获取输入的文本区。ROWS 和 COLS 负责规定它的大小

16.6.2 创建 HTML 表单

含有文本字段、单选按钮、复选框和弹出菜单的简单表单。首先通过一个简单文档以及产生它的 HTML 代码来看一下用户输入是如何进入浏览器的。用户可以点击按钮，或者在文本框中填入数据。这个示例将不处理输入的内容，因此当用户按下 Submit 按钮时，程序会将一个出错消息发送给服务器的错误日志。浏览器不显示任何内容。HTML 文件一般都保存在服务器根目录下的 htdocs 子目录中。如果 HTML 文件是在本机上创建的话，则可在地址栏中使用 file:/// 协议，该协议含有 HTML 文件的全路径，一般都带有扩展名 .html 或 .htm。

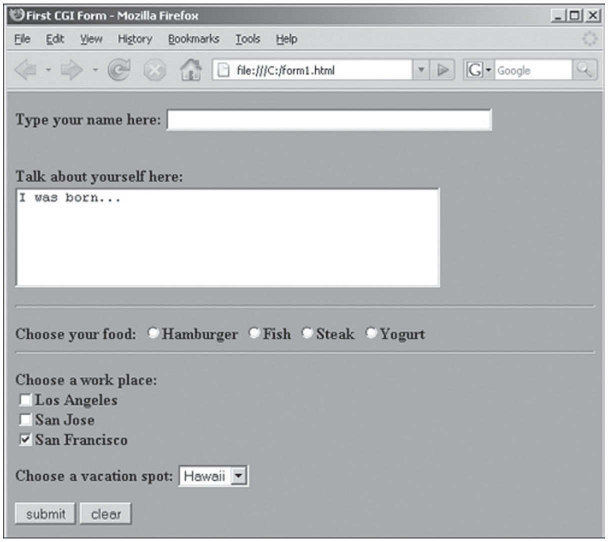


图 16-10　最初显示的表单

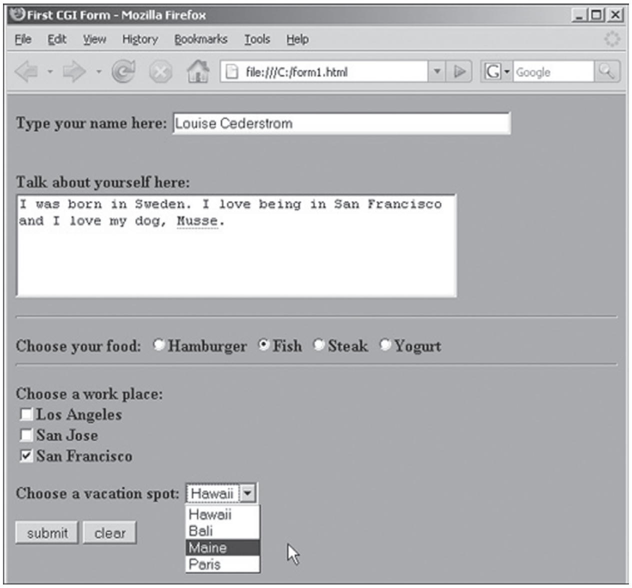


图 16-11　填入用户输入的表单

示例 16.11

```
    (The HTML Form Source File)
1    <html><head>
2    <title>First CGI Form</title></head>

     <body bgcolor="#9999cc">
3    <form action="/cgi-bin/bookstuff/form1.cgi" >
     <p/>
4     <b> Type your name here:
```

```
5       <input type="text" name="namestring" size=50>
        <p><br>Talk about yourself here: <br>
7       <textarea name="comments" rows=5 cols=50>I was born...
    </textarea>
    </b>
     <hr>
8   </p><b> Choose your food:
9   <input type="radio" name="choice" value="burger"/>Hamburger
    <input type="radio" name="choice" value="fish"/>Fish
    <input type="radio" name="choice" value="steak"/>Steak
    <input type="radio" name="choice" value="yogurt"/>Yogurt
    <hr>
    </p> <b>Choose a work place:</b> <br>

10  <input type="checkbox" name="place" value="la"/>Los Angeles
    <br>
    <input type="checkbox" name="place" value="sj"/>San Jose
    <br
    <input type="checkbox" name="place" value="sf" checked/>San Francisco
    </p>
    <b>Choose a vacation spot:</b>
11  <select name="location"> <option selected value="hawaii"/> Hawaii
    <option value="bali"/>Bali
    <option value="maine"/>Maine
    <option value="paris"/>Paris
    </select> <p>

    <input type="submit" value="submit">
    <input type="reset" value="clear">
    </body>
    </form>
    </html>
```

解释

1. 该标签说明 HTML 文档的开始。
2. *<TITLE>* 标签：标题出现在浏览器的主窗口外面。
3. *<FORM>* 标签的起始位置，规定浏览器在何处发送输入数据，以及如何处理输入数据。默认的方法是 GET。在提交数据时，服务器会执行 CGI 脚本。CGI 脚本位于服务器根目录下的 cgi-bin 目录中，CGI 脚本一般都保存在这里。在本例中，cgi 脚本位于 bookstuff 下的 cgi-bin 目录中。
4. *<P>* 标签表明开始新段。*<B>* 标签表明随后的文本是粗体。要求用户提供输入。
5. 输入类型是一个可放 50 个字符的文本框。当用户在文本框中输入文本时，文本内容就保存在用户定义的 *NAME* 值 namestring 中。例如，如果用户输入 Louise Cederstrom，则浏览器会将字符串 namestring=Louise Cederstrom 赋予查询字符串。如果指定了 *VALUE* 属性，则文本字段也可取默认值；即，浏览器刚显示它时出现在文本框中的文本。
6. 要求用户输入。
7. 文本区（text area）类似于文本字段（text field），但它允许扫描多行输入内容。<textarea> 标签会生成一个方框（名叫 comments），其大小由行数和列数决定（5 行乘以 15 列），并带有一个可选的默认值（I was born...）。
8. 要求用户从一系列菜单中选择。
9. 第一个输入控件的类型是一组单选按钮。用户只能从中选择一个按钮。该输入类型拥有两个属性：type 和 name。例如，如果用户选择了 Hamburger 这一项的话，则会把 name 的 choice

属性赋值为 burger，并将 choice=burger 传递给 CGI 程序。如果用户选择了 Fish，则会把 choice=fish 赋予查询字符串，依此类推。在点击 Submit 按钮后，这些键 / 值对将用于传递给 CGI 程序的查询字符串中。

10. 这次的输入类型是复选框。用户可以同时选择多个复选框。这里已经选定了其可选的默认值。当用户选择某个复选框时，会将 VALUE 属性的一个值赋予 NAME 属性。例如，如果选择了 Los Angles，则 place ＝ LA。
11. <select> 标签用于生成弹出菜单（又称作下拉列表）或滚动列表。它必须提供 name 选项，用于定义选项组的名字。对于弹出菜单而言，size 属性并不是必需的，其默认值为 1。弹出菜单最初只显示一项，在单击该项时才会展开菜单。用户只能在菜单中选择一项内容。如果给出了 size 属性，则可同时显示多个选项。如果指定了 multiple 属性（例如 select multiple name=whatever），则菜单会成为滚动列表的形式，显示其所有选项。
12. 如果用户单击 Submit 按钮，就会运行表单 ACTION 属性中列出的 CGI 脚本。在本例中并没有编写该脚本程序，因此会将出错信息发送到服务器的错误日志中和浏览器上。
13. 如果单击 Clear 按钮，便会把所有的输入框重置为其默认值。

16.6.3 GET 方法

用户可以通过称作 GET 的方法来创建类型最为简单的表单。每当浏览器请求文档时，都会用到这个方法。如果没有指定其他方法的话，GET 方法就是默认的方法。它是用于检索静态 HTML 文件与图像的惟一方法。

HTML 是一种面向对象的语言，读者应当知道，面向对象的子例程又称作方法。为了把数据传递给 CGI 程序，GET 方法会将输入追加到程序的 URL 中，这些输入往往是一个 URL 编码的字符串。环境变量 QUERY_STRING 将赋值为这种编码字符串，正如示例 16.12 所示。

服务器往往会对 URL 的长度有所限制。例如，UNIX 的长度限制为 1024 字节。如果需要向服务器发送更多的信息，则应使用 POST 方法。

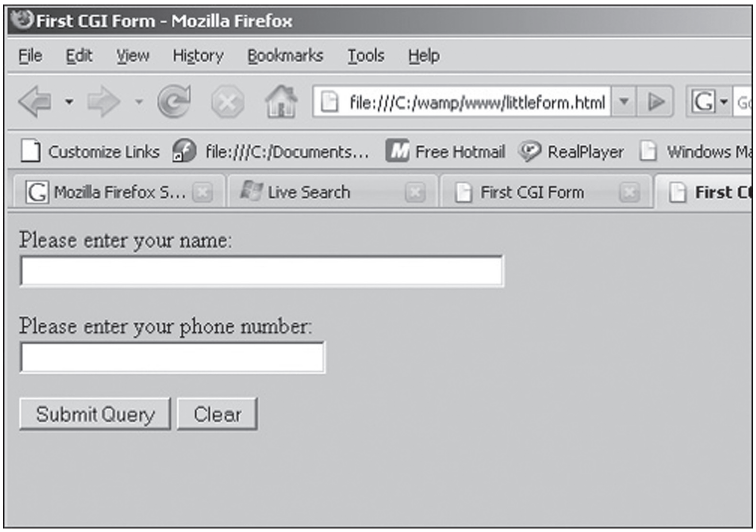


图 16-12　示例 16.12 中创建的 HTML 表单

示例 16.12

```
HTML Source File with a Form Tag and ACTION Attribute
----------------------------------------------------------------------
    <html>
      <head><title>First CGI Form</title></head>
    <body bgcolor=silver>
1     <form action="/cgi-bin/littleform.cgi" method=get>
          <!-- When user presses "submit", cgi script is called to
           process input -->
2          Please enter your name: <br>
3          <input type="text" size=50 name="Name">
           <p/>
       Please enter your phone number: <br>
4      <input type="text" size=30 name="Phone">
         <p/>
5      <input type=submit>
       <input type=reset value="Clear">
6    </form>
    </body>
```

解释

1. 表单创建在 HTML 文档的 <body> 标签内。<form> 标签规定了 URL 和用于处理表单的方法。当用户提交表单时，浏览器会将它获得的所有数据发送到 Web 服务器。action 属性告诉服务器在 URL 指定的位置调用 CGI 脚本，并将所有数据传递给该程序进行处理。method 属性告诉浏览器如何把输入的数据发送到服务器。GET 方法是其默认的方法，因此无需在这里指定。CGI 程序可以对数据进行任何必要的处理，并在完成后将其发送回服务器。然后由服务器将信息反馈给浏览器显示出来。
2. 要求用户提供输入。
3. 输入类型可以是一个持有 50 个字符的文本框。将 NAME 属性赋值为 Name。这是键 / 值对的键。用户应在文本框中输入一些内容。用户输入的值将赋予 NAME 键。浏览器将以这种格式把 name=value 键值对发送给 CGI 脚本，譬如 Name=Christian。
4. 输入类型的 name 属性为 Phone。浏览器会把用户在文本框中的输入内容以 Phone=value 的

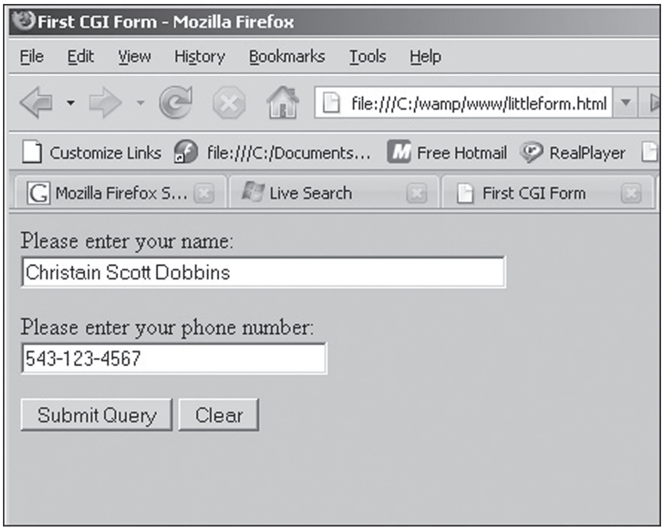


图 16-13　填充示例 16.13 提供的表单

格式发送给 CGI 程序。例如，Phone=543-123-4567。

5. 输入类型中的 submit 属性使得出现一个 Submit 按钮，按钮上默认写着字符串 Send Query。一旦用户选择该按钮框，就会执行 CGI 程序。reset 属性使得用户能够通过点击 Clear 按钮清空所有的输入设备。
6. </form> 标记终止该表单。

CGI 脚本

示例 16.13

```
(The CGI Script)
1  #!c:/ActivePerl/bin/perl.exe
   # The CGI script that will process the form information sent
   # from the server

2  print "Content-type: text/html\n\n";

print "Processing CGI form :\n\n";
# Print out only the QUERY_STRING environment variable

while(($key, $value)=each(%ENV)){
  print "<h3>$key = <em>$value</em></h3><br>"
  if $key eq "QUERY_STRING";
}
```

解释

1. shbang 行告诉服务器在哪里可以找到 Perl 解释器。它必须位于程序最顶部的第一行，并含有指向 CGI 脚本的精确路径。该行如果出错，就会造成“Internal Server Error”错误。在调试程序时，请读者牢记检查该行内容。
2. Perl 的输出内容会发送到浏览器而不是屏幕上。其内容类型（又称作 MIME 首部）是 text/html，因为其文本中含有 HTML 标签。
3. Perl 的输入来自于服务器。使用 while 循环迭代遍历 %ENV 散列中的所有环境变量。这些变量将从 Web 服务器传入 Perl 脚本中。
4. 只有找到环境变量 QUERY_STRING 的值时，才会打印该行内容。不过这并不需要迭代遍历整个列表。用户只需键入 print "$ENV{QUERY_STRING}
" 就足够了。

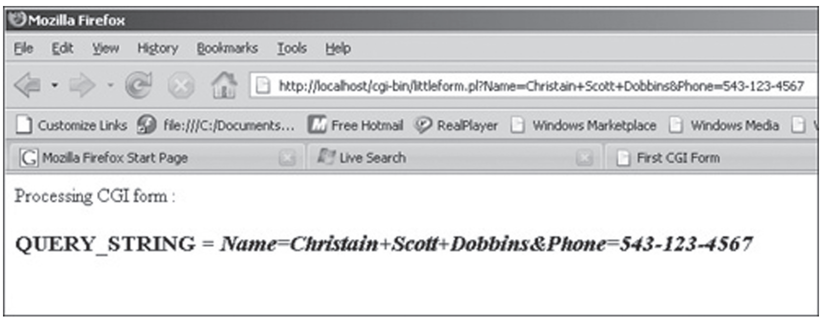


图 16-14　CGI 脚本输出

16.6.4 处理编码数据

编码查询字符串。 当使用 GET 方法时，会将信息放在环境变量 QUERY_STRING 中发送给 CGI 程序[⊖]。该字符串是一段 URL 编码。事实上，位于 HTML 表单中的所有数据都是以这种编码格式从浏览器发送到服务器的。在使用 GET 方法时，读者可以在浏览器上看到这些编码字符串，在它前面有一个问号。位于问号？后面的字符串将在环境变量 QUERY_STRING 中发送给 CGI 程序。其每个键 / 值对都由 & 符号隔开，而空格则以加号（+）来代替。此外还应当把所有非字母和数字的字符值替换为与之相等的十六进制值，并在其前面加上百分号（%）。在前面的示例中，当用户按下 Submit 按钮之后，便可在浏览器的 Location 框（Netscape）中观察到输入字符串。这些输入字符串已经附加到 URL 上，其前面带有一个问号。下面示例中的突出部分则是赋值给环境变量 QUERY_STRING 的内容。环境变量 QUERY_STRING 将在散列 %ENV 中传递给 Perl 脚本。如要访问 Perl 脚本中的键 / 值对，读者不妨添加一个 print 语句：print $ENV{QUERY_STRING};。

示例 16.14

```
1   What you see in the Location box of the browser:

http://servername/cgi-bin/
form1.cgi?Name=Christian+Dobbins&Phone=543-123-4567

2   What the server sends to the browser in the ENV hash value,
QUERY_STRING:

QUERY_STRING=Name=Christian+Dobbins&Phone=543-123-4567
```

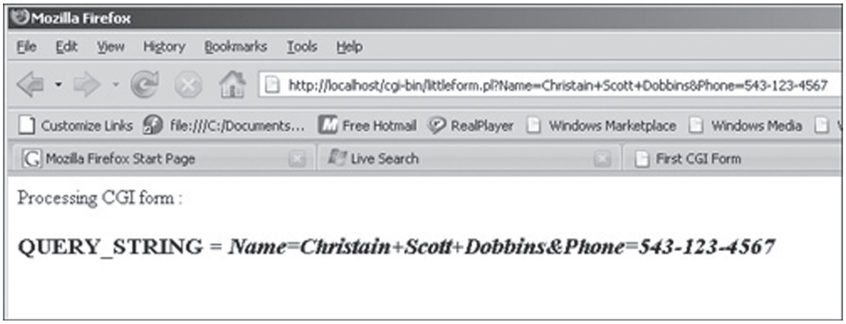


图 16-15 示例 16.14 中 CGI 脚本的输出

借助 Perl 解码查询字符串。 对查询字符串进行解码并不困难，因为 Perl 已经提供了很多字符串操作函数，如 tr、s、split、substr、pack 等等。当用户从服务器获得查询字符串并将它传入 Perl 程序之后，便可对它进行解析，并随心所欲地处理所得数据。为了从查询字符串中剔除 &、+ 和 = 符号，可使用替换命令 s、split 函数或翻译函数 tr。为了对前面带有 % 符号的字符进行从十六进制值到字符的转换，一般可使用 pack 函数。表 16-8 和表 16-9 列举了查询字符串中的编码符号和 URL 十六进制编码字符。

⊖ 当使用 POST 方法时，会把输入从 STDIN 赋值给一个变量，并以同样的方式编码。

表 16-8　查询字符串中的编码符号

符号	功　能
&	分隔键 / 值对
+	替换空格
%xy	代表其十六进制值小于 21（即十进制 33）或大于 7f（即十进制 127）的 ASCII 字符和特殊字符？、&、%、+ 和 =。这些字符的开头必须出现一个 %，其后是与该字符值相等的十六进制值。譬如，%2F 代表正斜杠，%2c 则代表逗号

表 16-9　URL 十六进制编码字符

字符	值	字符	值
制表符（Tab）	%09	<	%3C
空格（Space）	%20	=	%3D
!	%21	>	%3E
"	%22	?	%3F
#	%23	@	%40
$	%24	[	%5B
%	%25	\	%5C
&	%26	]	%5D
(	%28	^	%5E
)	%29	'	%60
,	%2C	{	%7B
.	%2E	|	%7C
/	%2F	}	%7D
:	%3A	~	%7E
;	%3B		

使用 Perl 解析表单输入。当 Perl CGI 脚本从表单获得输入内容后，就对它进行解码。在该过程中，脚本会负责拆分其键 / 值对，并以常规文本替换其中的特殊字符。解析完毕后，便可使用这些信息来创建客户登记本、数据、向用户发送电子邮件等等。

用于解析编码字符串的例程可以放在子例程中，并保存到读者自己的库里面。或者也可利用 CGI.pm 库模块进行解析。该模块是 Perl 标准发布版的一部分，它避免了所有的麻烦。

解码查询字符串。解码的各个步骤均由 Perl 函数进行处理。下面显示了一个赋值给 $ENV{QUERY_STRING} 的 URL 编码字符串。

```
Name=Christian+Dobbins&Phone=543-456-1234
```

其键 / 值对表明该 URL 由三段信息组成：Name、Phone 和 Sign，这些信息都由 & 号隔开。

```
Name=Christian+Dobbins&Phone=543-456-1234
```

首先要做的是拆分该行信息并创建数组（详见下面的步骤 1）。当基于 & 拆分完毕字符串后，便可使用 tr 或 s 函数删除 + 符号。然后，以 = 作为拆分定界符（详见步骤 2），使用 split 函数把其余的字符串继续拆分为键 / 值对。

```
1.  @key_value = split(/&/, $ENV{QUERY_STRING});
    print "@key_value\n";

2.  Output:
    Name=Christian+Dobbins  Phone=543-456-1234
```

拆分查询字符串，创建 @key_value 数组。

```
3.  foreach $pair ( @key_value){
        $pair =~ tr/+/ /;
        ($key, $value) = split(/=/, $pair);
        print "\t$key: $value\n";
     }
```

Name=Christian+Dobbins	Phone=543-456-1234

```
4.  Output:
    Name: Christian Dobbins
    Phone: 543-456-1234
```

示例 16.15

```
(Another URL-Encoded String assigned to $ENV{QUERY_STRING})

1   $input="string=Joe+Smith%3A%2450%2c000%3A02%2F03%2F77";
2   $input=~s/%(..)/pack("c", hex($1))/ge;
3   print $input,"\n";

    Output:
    string=Joe+Smith:$50,000:02/03/77
```

解释

1. 该字符串中含有小于十进制 33 和大于 127 的 ASCII 字符、冒号、美元符号、逗号和正斜杠。
2. 使用 pack 函数将十六进制编码格式的字符转换回字符格式。
3. 替换操作的搜索端 /%(..)/ 是一个正则表达式，含有百分号实量和位于括号中的两个字符（其中每个点号代表一个字符）。这里使用了小括号，以便让 Perl 将找到的两个字符保存到特殊标量 $1 中。

在替换端，首先使用 pack 函数将保存在 $1 中的两个十六进制字符值转换为其相应的十进制值，然后再把得到的十进制值转换为无符号的字符。最后把结果赋值给标量 $input。

现在则必须删除 + 符号。

16.6.5 总结

GET 方法。现在可以总结一下 Perl CGI 通过 GET 方法可以处理的表单。当用户填充了表单并按下 Submit 按钮后，CGI 会解码查询字符串，并将最终结果显示到返回的 HTML 页面上。

下面的示例将展示：

1. 填充好的 HTML 表单。
2. 生成表单的 HTML 源文件。
3. CGI 处理后的表单。
4. 负责处理表单的 Perl CGI 脚本。

示例 16.16

```
(The HTML source file)
    <html><head><title>cgi form</title></head><body>
    <hr/>
1   <form action="http://127.0.0.1/cgi-bin/getmethod.cgi" method=get>
    <!When user presses "submit", cgi script is called to process input >
2   Please enter your name: <BR>
3   <input type="text" size=50 name=name>
    <P/>
    Plese enter your salary ($####.##): <BR>
    <INPUT TYPE="text" SIZE=30 NAME=Salary>
    <P/>
    Plese enter your birth date (mm/dd/yy): <BR>
    <input type="text" size=30 name=birthdate>
    <P/>
4   <input type=submit value="Submit Query">
    <INPUT TYPE=RESET VALUE="Reset">
5   </form>
    </body></html>
```

解释

1. 表单是从 <form> 标签开始的。用户在表单上点击 Submit 按钮后，action 属性会触发本机（本地主机 IP 地址为 127.0.0.1）上的 HTTP 服务器启动其根目录下 cgi-bin 目录中的 getmethod.cgi 脚本。
2. 要求用户提供输入信息。
3. 用户在文本框中填入自己的名字、薪金等。
4. 当用户按下 Submit 按钮时，激活 action 属性指定的 CGI 脚本。
5. 这里是表单标签的结束位置。

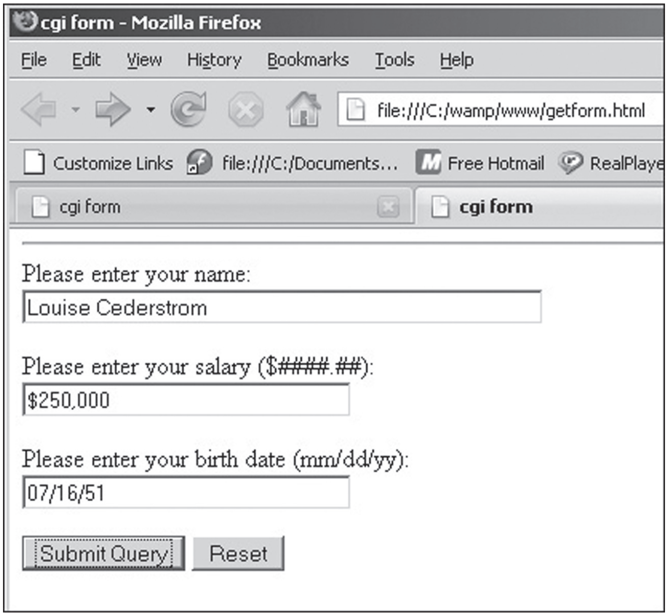


图 16-16　示例 16.16 中的 HTML 表单

示例 16.17

```
#!c:/perl/bin/perl
# The CGI script that processes the form shown in Example 8.12.
1 print "Content-type: text/html\n\n";
2 print <<HTML;
 <html><title>Decoding the Input Data</title>
 <body bgcolor="lavender">
 <font face="verdana" size="+1">
 HTML
 print "<H3><U>Decoding the query string</U></H3>";
 # Getting the input
3 $inputstring=$ENV{QUERY_STRING}};
  print "<B>Before decoding:</B>";
  print "<P>$inputstring";
  # Extracting the + and & and creating key/value pairs
4 @key_value=split(/&/,$inputstring);
  foreach $pair ( @key_value){
5    ($key, $value) = split(/=/, $pair);
6    $value=~s/%(..)/pack("C", hex($1))/ge;
     $value =~ s/\n/ /g;
 $value =~ s/\r//g;
 $value =~ s/\cM//g;
7    $input{$key}=$value ; # Creating a hash
  }
  # After decoding
  print "<HR>";
  print "<P><B>After decoding:</B><P>";
8    while(($key, $value)=each(%input)){
      print "$key: <I>$value</I><BR>";
  }
  print <<HTML;
   <hr>
   <p>Now what do we want to do with this information? </p>
   </font>
   </body>
   </html>
  HTML
```

解释

1. MIME 首部行表明由该程序返回的数据格式为 HTML 文本。在首部信息后面必须提供两个换行符（这是必需的！）。
2. 散列 %ENV 中含有 Web 服务器发送给该 Perl 程序的键 / 值对。将环境变量 QUERY_STRING 的值赋予标量 $inputstring。
3. tr 函数将所有的 + 转换为空格。
4. pack 函数将十六进制数字转换为其对应的 ASCII 字符。
5. 将标量 $inputstring 的值从 Perl 脚本发送给服务器，然后再返回给浏览器。
6. 依据 & 符号拆分标量 $inputstring。将获得的键 / 值对保存到含有三个元素的数组 @key_value 中，其内容是：

   Name=Louise Cederstrom&Salary=$200,000&Birthday=7/16/51
7. 使用 foreach 循环迭代遍历数组 @key_value。以 = 符号拆分每个数组元素，创建并返回键 / 值对。
8. 使用相应的键 / 值对创建一个新散列 %input。
9. 使用 while 循环迭代遍历散列。

10. 打印新的键值对，并将它发送回 Web 服务器。浏览器将显示这些输出内容。当 Perl 脚本解析并保存好来自表单的输入后，程序员可以自由决定如何使用这些数据。他可以向用户回复电子邮件，将数据保存到数据库中，或者创建地址簿，等等。真正的工作就是这些！

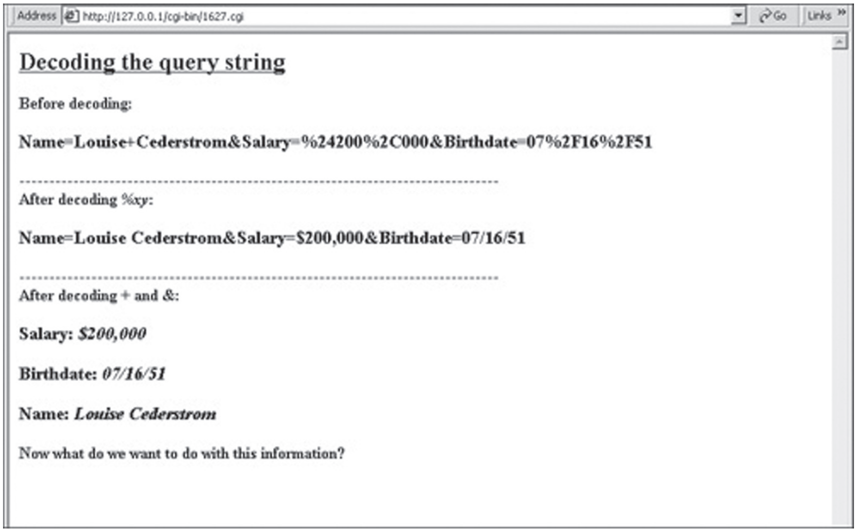


图 16-17　示例 16.17 中 CGI/Perl 处理后的输出

16.6.6　POST 方法

GET 方法和 POST 方法之间的惟一区别是二者从服务器把输入传送到 CGI 程序的方式。当使用 GET 方法时，服务器将把输入放入环境变量 QUERY_STRING 中，然后发送给 CGI 程序。

当使用 POST 方法时，CGI 程序将从标准输入 STDIN 中获取输入。这两种方式的输入编码格式完全相同。使用 POST 方法的原因之一是有的浏览器限制了保存到环境变量 QUERY_STRING 中的数据总量。而 POST 方法是不会将其数据存储到查询字符串中的。另外，GET 方法会在浏览器 Location 框内的 URL 中显示所有输入数据，而 POST 方法则能隐藏这些数据。由于 POST 方法不会把输入附加到 URL 上，因此常用于处理那些需要填入很多数据的表单。

在 HTML 文档中，<FORM> 标签表明表单的起始位置。ACTION 属性告诉浏览器将用户选定的数据发送到哪里，Method 方法则告诉浏览器如何去发送。如果使用 POST 方法的话，会把浏览器的输出发送到服务器，然后再到 CGI 程序的标准输入 STDIN。数据总量将保存在环境变量 CONTENT_LENGTH 中，表示用户提供的输入内容字节数。

浏览器不会把输入内容赋予环境变量 QUERY_STRING，而是在消息体中发送到服务器的，这和电子邮件的发送方式较为类似。然后由服务器负责填充所有数据，并转发给 CGI 程序。

Perl 的 read 函数会读取 CONTENT_LENGTH 字节数，将输入保存到标量中，然后通过与处理查询字符串相同的方法来处理它。这里改变的并不是输入的格式，而是进入程序的方式。请注意，在使用 POST 方法后，浏览器的 Location 框中不再像 GET 方法时那样在 URL 中含有输入内容了。

示例 16.18

```
     (The HTML source file)
     <HTML>
     <HEAD>
     <TITLE>CGI Form</TITLE>
     <HR>
1    <FORM ACTION="http://127.0.0.1/cgi-bin/postmethod.cgi" METHOD=POST>
     <!When user presses "submit", cgi script is called to process input >
2    Please enter your name: <BR>
3    <INPUT TYPE="text" SIZE=50 NAME=Name>
     <P>
     Please enter your salary ($####.##): <BR>
     <INPUT TYPE="text" SIZE=30 NAME=Salary>
     <P>
     Please enter your birth date (mm/dd/yy): <BR>
     <INPUT TYPE="text" SIZE=30 NAME=Birthdate>
     <P>
4    <INPUT TYPE=SUBMIT VALUE="Submit Query">
     <INPUT TYPE=RESET VALUE="Reset">
5    </FORM>
     </HTML>
```

解释

1. <FORM>标签表示表单在这里开始。将 ACTION 属性赋值为 CGI 脚本 postmethod.cgi 的 URL。该脚本将在用户点击 Submit 按钮时执行。这里还将 METHOD 属性赋值为 POST，表明如何处理来自表单的数据。
2. 要求用户输入。
3. 创建文本字段（Text field），用于保存用户姓名、薪金和生日。
4. 创建 Submit 按钮。
5. 结束表单。

示例 16.19

```
#!c:/perl/bin/perl
# The CGI script that processes the form shown in Example 8.12.
1 print "Content-type: text/html\n\n";
2 print <<HTML;
 <html><title>Decoding the Input Data</title>
 <body bgcolor="lavender">
 <font face="verdana" size="+1">
 HTML
 print "<H3><U>Decoding the query string</U></H3>";
 # Getting the input
3 $inputstring=$ENV{QUERY_STRING}};
  print "<B>Before decoding:</B>";
  print "<P>$inputstring";
  # Extracting the + and & and creating key/value pairs
4 @key_value=split(/&/,$inputstring);
  foreach $pair ( @key_value){
5      ($key, $value) = split(/=/, $pair);
6      $value=~s/%(..)/pack("C", hex($1))/ge;
       $value =~ s/\n/ /g;
       $value =~ s/\r//g;
       $value =~ s/\cM//g;
7      $input{$key}=$value ; # Creating a hash
  }
  # After decoding
  print "<HR>";
```

```
    print "<P><B>After decoding:</B><P>";
8   while(($key, $value)=each(%input)){
        print "$key: <I>$value</I><BR>";
    }
    print <<HTML;
    <hr>
    <p>Now what do we want to do with this information? </p>
    </font>
    </body>
    </html>
    HTML
```

解释

1. 这个 MIME 首部行表明，该程序返回的数据格式将是一段 HTML 文本。在首部信息后面必须提供两个换行符（这是必需的！）。
2. here 文档用于生成 HTML 页面的开头部分。
3. %ENV 散列中含有 Web 服务器发送给 Perl 程序的键 / 值对。环境变量 QUERY_STRING 的内容将会赋值给标量 $inputstring。
4. 根据 & 符号拆分标量 $inputstring。将其返回值保存到一个含有三个元素的数组 @key_value 中，形如：

   Name=Louise+Cederstrom Salary=$200,000 Birthdate=7/16/51

   $key_value[0] $key_value[1] $key_value[2]
5. foreach 循环会逐一读取数组中的每个元素。首先将每个元素通过“=”号拆分，从而生成两个值，第一个值名叫 $key，表示来自 HTML 表单的输入设备名称，如“Name”；第二个值名为 $value，含有用户键入到输入设备中的数据，譬如“Louise+Caderstorm”。
6. 现在通过 Perl 的 pack 函数将十六进制值解码为各自对应的 ASCII 字符值。
7. 通过加码键 / 值对，创建一个新的散列 %input。
8. 使用 while 循环迭代遍历散列。打印新的键值对，并将它们发送回 Web 服务器。浏览器将显示这些输出。当 Perl 脚本解析并保存好来自表单的输入后，程序员可以自由决定如何使用这些数据。他可以向用户回复电子邮件，将数据保存到数据库中，或者创建地址簿，等等。真正的工作就是这些！

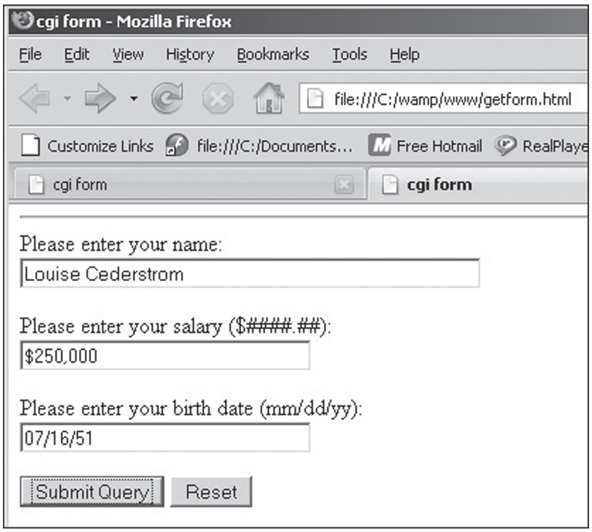


图 16-18 示例 16.18 中的 HTML 输入表单

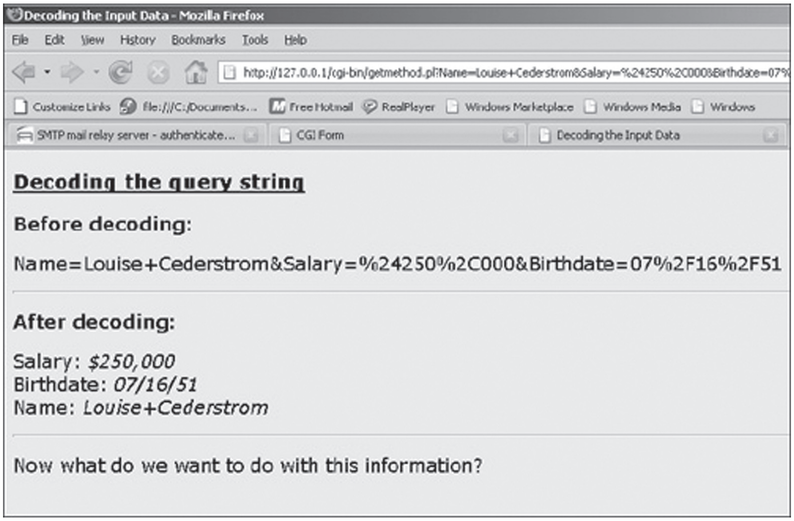


图 16-19　示例 16.19 中 CGI 脚本的输出

16.6.7　处理电子邮件

SMTP 服务器。 在处理表单时，经常需要在退出前发送电子邮件。读者可以通过电子邮件将提交的表单数据发送给用户和 / 或自己。如果没有有效的 SMTP（简单邮件传输协议）服务器[⊖]的话，就不能在 Internet 上发送电子邮件。

SMTP 服务器是一种邮件 daemon 程序的实例，它能在端口 25 处监听到来的邮件。SMTP 是一种基于 TCP 协议的客户端 / 服务器协议，其客户端发送消息到服务器。UNIX 系统一般都使用一种名叫 sendmail 的邮件程序来作为 SMTP 服务器，负责监听到达的邮件。读者通常可以在命令行中运行 sendmail，并以接收者的名字作为参数。如要终止一个电子邮件消息，可把一个点号放在单独一行即可。CGI 脚本中是不能交互发送邮件的，因此读者可能需要通过 sendmail 的选项来控制这些特性。详见表 16-10。

表 16-10　sendmail 选项

选　项	用　途
-o	表示其后是 sendmail 选项
-t	从消息体中读取 To、From、Cc 和 Bcc 首部信息
-f "email address"	消息来自这个电子邮件地址
-F "name"	消息来自这个人名
-i	如果在行内，忽略其点号
-odq	将多个电子邮件消息放入队列，以便实现异步传递

对于 Windwos，也有两个类似于 sendmail 的程序。首先是 Blat，它是一个得到广泛使用的公共

⊖ Internet 邮件消息的格式由 RFC822 定义。

Win32 控制台实用程序，可通过 SMTP 协议发送电子邮件（详见 www.interlog.com/~tcharron/blat.html）。另一个是 wSendmail，这是一种小型的电子邮件实用程序，可以从程序、命令行或直接在 HTML 表单中发送电子邮件（详见 http://www.scriptarchive.com/Internet/E_Mail/E_Mail_Tools 或 http://www.developertutorials.com/tutorial/cgi-Perl/email-with-Perl-2/page4.html）。读者亦可访问 CPAN，并从中搜索 MailFolder 包。该包含有诸如 Mail::Folder、Mail::Internet 和 Net::SMTP 之类的模块，从而可以进一步简化邮件的发送和接收工作。

示例 16.20

```
(From the HTML form where the e-mail information is collected)
<html>
<head><title>Register Now!</title></head>
<body bgcolor="ccff66">
<form method="post" action="http://127.0.0.1/cgi-bin/mailer.pl">
<input type="hidden" name="xemailx"
        value="elizabeth@ellieq.com">
<input type="hidden" name="xsubjext"
        value="course registration">
<input type="hidden"  name="xgoodbyex"
        value="thank you for registering.">
<p>
<a name="registration">
<table cellspacing=5 cellpadding=3>

<tr> <td align=right><b>First name:</b></td>
    <td align=left><input type=text size=30 name="first_name*"
        value=""></td>
</tr>
</p>
<tr> <td align=right><B>Last Name:</B></TD>
    <td align=left><input type=text size=30 name="last_name*"
        value=""></td>
</tr>
<tr>
    <td align=right><b>Company:</b></td>
    <td align=left><input type=text size=30 name="company*"></td>
</tr>
<tr>
    <td align=right><b>Address:</b></td>
    <td align=left><input type=text size=30 name="address1*"
value=""></td>
</tr>
<tr>
    <td align=right><b>City/Town:</b></td>
    <td align=left><input type=text size=30 name="city*"
        value=""></td>
</tr>
<tr>
    <td align=right><b>State/Province:</b></td>
    <td align=left>
        <select name="state" size="1">
            <option value="AL">Alabama</option>
            <option value="AK">Alaska</option>
            <option value="AZ">Arizona</option>
```

```
                <option value="AR">Arkansas</option>
                <option value="CA">California</option>
                <option value="CO">Colorado</option>
                <option value="CT">Connecticut</option>
                     --- continue here ---
                <option value="WA">Washington</option>
                <option value="WV">West Virginia</option>
                <option value="WI">Wisconsin</option>
                <option value="WY">Wyoming</option>
            </select>
        <td align=right><b>Postal/Zip code:</b></td>
        <td align=left><input type=text size=10 name="zip"
             value=""></td>
</tr></b></td>
</table>
<p>
    <input type=submit value="Register now!"></td>
    <input type=reset></td>
</tr>
</form>
</body>
</html>
```

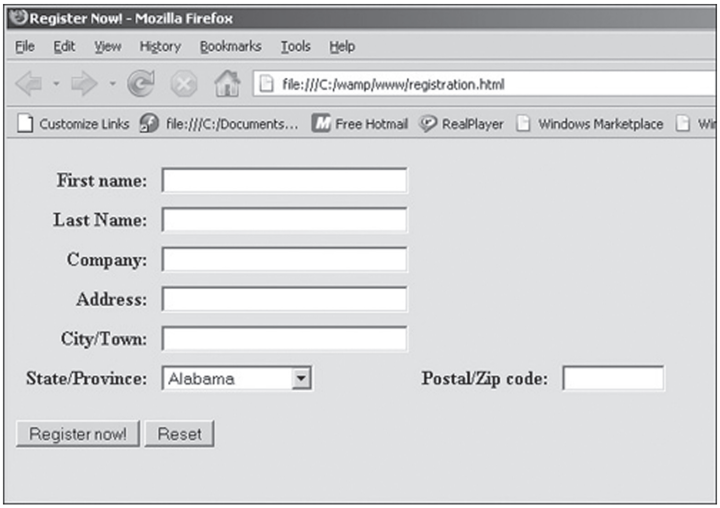


图 16-20 示例 16.20 中的部分 HTML 注册表单

示例 16.21

```
(CGI script to Handle email--only a portion of the script)
# An HTML Form was first created and processed to get the name of the
# user who will receive the e-mail, the person it's from, and the
# subject line.
    $mailprogram="/usr/lib/sendmail";  # Your mail program goes here
    $sendto="$input{xemailx}";         # Mailing address goes here
    $from="$input{xmailx}";
    $subject="$input{xsubjext}";
```

```
5   open(MAIL, "|$mailprogram -t  -oi") || die "Can't open mail
        program: $!\n";
    # -t option takes the headers from the lines following the mail
    # command -oi options prevent a period at the beginning of  a
    # line from meaning end of input
6   print MAIL "To: $sendto\n";
    print MAIL "From: $from\n";
    print MAIL "Subject: $subject\n\n";

7   print MAIL <<EOF;    # Start a "here document"

    Registration Information for $input{$first_name}
        $input{$last_name}:
    Date of Registration: $today
    ----------------------------------------------
    First Name:            $input{$first_name}
    Last Name:             $input{$last_name}
    Street Address:        $input{$address}
    City:                  $input{$city}
    State/Province:        $input{$state}

    <Rest of message goes here>

8   EOF
9   close MAIL;  # Close the filter
```

解释

1. 本例使用的电子邮件程序名为 sendmail，位于 UNIX 的 /usr/lib 子目录中。
2. 将本行内容赋值给电子邮件中的 To: 首部。
3. 将本行内容赋值给电子邮件中的 From: 首部。
4. 本行内容是电子邮件中的 Subject: 首部。
5. Perl 准备打开 MAIL 过滤器，以便将用户的电子邮件消息发送到 sendmail 程序。-t 选项告诉 sendmail 在电子邮件文档中（而不是从命令行中）扫描 To:、From: 和 Subject: 行；-i 选项则告诉电子邮件程序忽略在行中发现的点号。
6. 这是首部行，负责说明邮件将发送给谁、从何处而来、以及邮件的主题。这些值均取自表单。
7. 开始 here 文档。将位于 EOF 和 EOF 之间的文本内容通过 MAIL 过滤器发送到 sendmail 程序。
8. EOF 标记表明 here 文档到此结束。
9. 关闭 MAIL 过滤器。

16.7　CGI.pm 模块

16.7.1　简介

在编写动态 CGI 程序（如客户端、页面计数器、反馈表单等）时，最为常用的 Perl 5 模块莫过于由 Lincoln Stein 编写的 CGI.pm 模块；该模块最开始出现在版本 5.004 的标准 Perl 库中。读者可以在 www.Perl.com/CPAN 处找到最新版本的 CGI.pm 模块。CGI.pm 不仅利用了 Perl 5 开始引入的面向对象特性，还提供了专门解释查询字符串的方法（GET 和 POST），并屏蔽了 HTML 语法细节。

Lincoln Stein 还撰写了 Official Guide to Programming with CGI.pm[1]（www.wiley.com/compbook/stein）。这是一个易于阅读的优秀指导文档，提供了如下信息。

16.7.2 优点

1. CGI.pm 允许程序员维护填好的表单（HTML）和负责解析它的脚本，它们都位于 cgi-bin 目录下的同一个文件中。通过这种方式，HTML 文件（含有表单）和 CGI 脚本（负责读取 / 解析处理表单数据）就不再分离了。[2]

2. 当用户填充了表单之后，结果将出现在同一页面上。换而言之，用户无需向后翻页查看表单上的内容，并且填好的表单也不会丢失数据，即维护了状态。这种数据不会消失的特性称作“粘性”。如要禁止该特性，请参阅“参数覆盖”一节。

3. 对表单数据的所有读取和解析工作都由模块负责处理。

4. 方法用于代替 HTML 标签，负责创建文本框、单选按钮、菜单等，以便创建表单或指定标准标签，如首部、标题、分段和水平标尺等。

5. 如要查看 CGI.pm 生成的 HTML 标签，可在表单显示之后在 View 菜单中选择 Source（Internet Explorer 中）。

6. 接受上传的文件，管理 cookie。这样使用 CGI.pm 模块会更容易。

16.7.3 使用 CGI.pm 编程的两种形式

面向对象的样式。 借助面向对象的样式，读者可以创建一个或多个 CGI 对象，并使用这些对象方法来创建页面的各个元素。每个 CGI 对象对以服务器传给 CGI 脚本的命令参数列表开头。用户可以修改这些对象，并将它们发送到文件或数据库中。每一个对象都是独立的，拥有自己的参数列表。当填写完表单后，其表单内容将可以从一个脚本运行维持到下一个脚本，即实现了状态的维护（一般 HTML 是无状态的，即在退出页面时所有的内容均会消失）。

示例 16.22

```
use CGI;
$obj=new CGI;  # Create the CGI object
print $obj->header,      # Use functions to create the HTML page
$obj->start_html("Object oriented syntax"),
$obj->h1("This is a test..."),
$obj->h2("This is a test..."),
$obj->h3("This is a test..."),
$obj->end_html;
```

解释

通过从浏览器窗口中查看源文件，便可看到如下输出。它表明了 CGI.pm 模块生成的 HTML 输出内容。本例中面向对象的 CGI 脚本输出内容如图 16-21 所示。

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

[1] Stein, L., Official Guide to Programming with CGI.pm: The Standard for Building Web Script, Wiley Computer Publishing, 1998.

[2] Ibid。

```
"DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
<head><title>Object oriented syntax</title></head><body>
<h1>This is a test...</h1>
<h2>This is a test...</h2>
<h3>This is a test...</h3>
</body
</html>
```

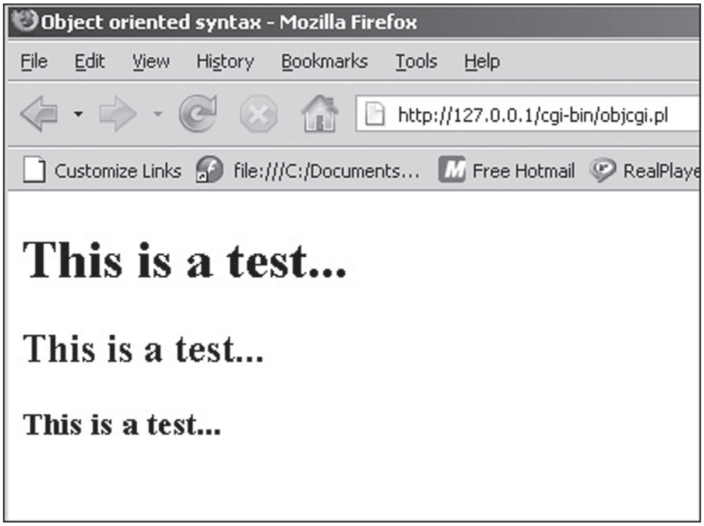


图 16-21 示例 16.22 中面向对象的 CGI 脚本输出内容

面向函数的样式。 面向函数的样式要比面向对象的样式更容易使用，因为它可以直接创建或操纵 CGI 对象。模块将创建一个默认的 CGI 对象。用户可以使用统一的内建函数来操作对象、向函数传递参数、创建 HTML 标签或者检索传入表单的信息。

尽管面向函数的样式能够提供更为简明的编程接口，但它一次只能使用一个 CGI 对象。

下面的示例使用了面向函数的接口。其中主要的区别在于必须将 :standard 函数导入到程序的命名空间中。读者无需为它创建 CGI 对象，它会自动创建[⊖]。示例 16.23 的输入内容如图 16-22 所示。

示例 16.23

```
#!/usr/bin/perl
use CGI qw(:standard);   # Function-oriented style uses a set of
                         # standard functions
print header,
  start_html("Function oriented syntax"),
  h1("This is a test..."),
  h2("This is a test..."),
  h3("This is a test..."),
end_html;
```

⊖ 默认创建的对象是 $CGI::Q。用户可以在需要时直接访问它。

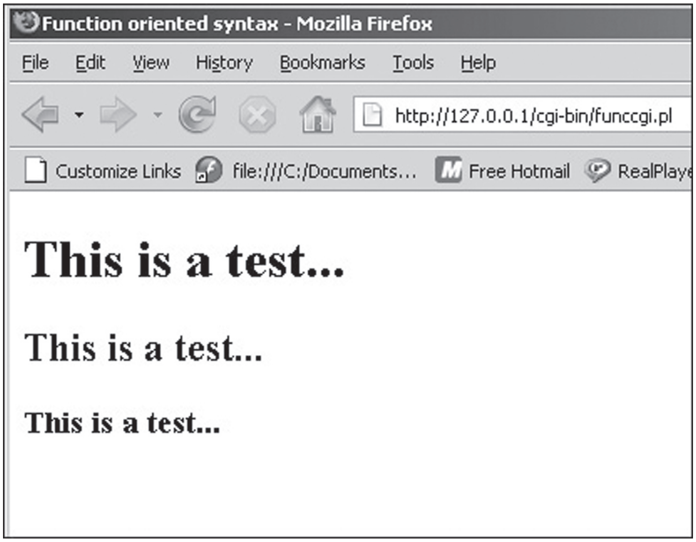


图 16-22　示例 16.23 中面向函数的 CGI 脚本输出内容

16.7.4　重要警示

Perl 的 print 函数用于将输出内容从脚本发送给服务器。不过在很多示例中，并不是对每一行分别使用不同的 print 语句，而是可以向 print 函数传递由逗号隔开的多个字符串。事实上，在 CGI.pm 文档中，类似于示例 16.23 的脚本会以一个 print 函数连续输出超过 30 行内容，并最终以一个分号结尾。这里很容易多写一个分号，并因此错误地提前结束输出。这样做会造成该程序中断执行，并

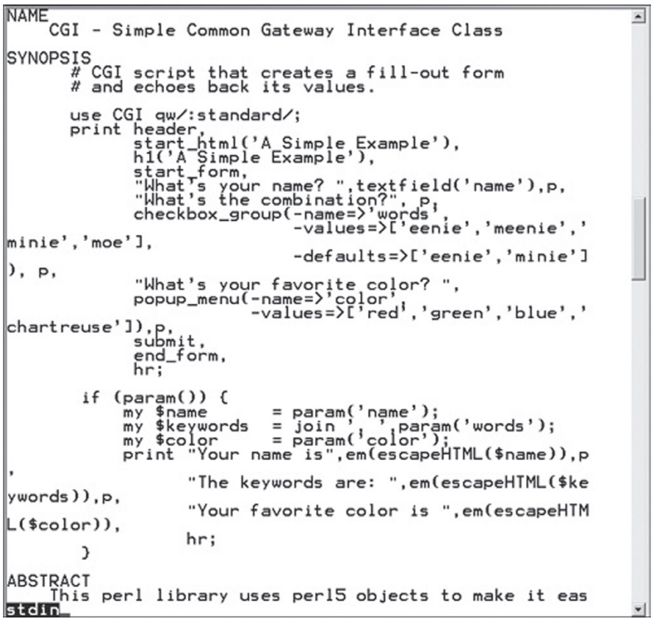


图 16-23　CGI.pm 文档页

发送一个“Internet Server Error”消息给浏览器。

另一个常见的错误是不恰当地终止了某个 here 文档。为了避免使用多行 print 语句，常常会用到 here 文档（详见第 4 章）。here 文档的终止字必须位于单独的行并紧靠其左侧边界，其两边不应出现空格，并以回车符结尾。

```
   #!/bin/perl
   # The HTML tags are embedded in the here document to avoid using
   # multiple print statements
1  print <<EOF;     # here document in a CGI script
2  Content-type: text/html
3
4  <HTML><HEAD><TITLE>Town Crier</TITLE></HEAD>
   <H1><CENTER>Hear ye, hear ye, Sir Richard cometh!!</CENTER></H1>
   </HTML>
5  EOF
   # The EOF in line 5 cannot have surrounding spaces and must be
   # terminated with a newline.
```

16.7.5　HTML 表单方法

一个 CGI 脚本由两个部分组成：一部分负责创建在浏览器中显示的表单；另一部分则负责从表单获取输入，解析输入，并通过将其发送回浏览器、数据库、电子邮件等方式来处理这些信息。

创建 HTML 表单。模块简化了 HTML 表单的创建流程。譬如，其中有的方法负责起始或终止 HTML 表单，有的则用于创建首部、复选框、弹出菜单、单选按钮、Submit 与 Reset 按钮等组件。表 16.11 列出了 GCI.pm 模块提供的最为常用的 HTML 方法。

当向 CGI.pm 中的方法传递参数时，可以使用两种形式：

命名参数（Named arguments）——以键 / 值对的形式传递，其参数名前面有短横线，并且是大小写敏感的。

示例 16.24

```
   (Named Arguments)
1  print popup_menu(-name=>'place',
                    -values=>['Hawaii','Europe','Mexico', 'Japan' ],
                    -default=>'Hawaii',
                   );
2  print popup_menu(-name=>'place',
                    -values=> \@countries,
                    -default=>'Hawaii',
                   );
```

解释

1. 要传递给 popup_menu 方法的参数又称为命名参数或参数列表。本例中的参数名分别是 -name、-value 和 -default。这些参数前面都带有短横线，并且是大小写敏感的。如果参数名可能与内建的 Perl 函数名或保留字冲突的话，就应当给参数加上引号。请注意，这里的参数是作为键 / 值对传递给方法的。-value 键对应的值是一个由国家组成的匿名数组。
2. 与前面一个示例完全相同，只不过其 -value 的值是指向国家数组的一个引用。在程序的其他位置创建数组 @contries 并给它赋值。

位置参数（Positional arguments）——以字符串形式传递，它们直接代表了值，用于简单的 HTML 标签。例如，CGI.pm 提供了 h1() 方法，负责生成 HTML 标签 <H1> 和 </H1>。h1() 的参数

则是插入到两个标签之间的文本字符串。其调用方法如下：

```
print h1("This is a positional argument ");
```

它会翻译为：

```
<H1> This is a positional argument </H1>
```

如果使用了 HTML 属性⊖，则其第一个参数是指向匿名散列的引用，该属性及其值都将在前导标签之前插入到列表中。例如：

```
print h1({-align=>CENTER} , "This heading is centered");
```

它会翻译为：

```
<H1 ALIGN="CENTER"> This heading is centered </H1>
```

如果参数是指向匿名列表的引用，则列表中的每一项都会以恰当的方式分配到各个标签中去。例如：

```
print li(['apples', 'pears', 'peaches']);
```

将翻译为三个列表项：

```
<LI>apples</LI><LI>pears</LI><LI>'peaches'</LI>
```

而：

```
print li('apples', 'pears', 'peaches');
```

则会翻译为：

```
<LI>apples pears peaches</LI>
```

示例 16.25

```
(The CGI script)
 #!c:/ActivePerl/bin/perl.exe
 # Shortcut calling styles with HTML methods

1  use CGI qw(:standard);  # Function-oriented style print header
2  print header;
# Note that the following functions are all embedded as a
# comma-separated list of arguments in one print statement.

3  print start_html(-title=>"Testing arguments",
                  -bgcolor=>"#99FF66"),
4     b(),font({-size=>"+2", -color=>"#006600"}),
5     p(),"\n",
6     p("This is a string"),"\n",
7     p({-align=>center}, "red", "green", "yellow"), "\n",
8      p({-align=>left}, ["red","green","yellow"]),
9      end_html;# Shortcut calling styles with HTML methods
```

解释

1. 使用 CGI.pm 模块的面向函数的样式。
2. header 函数会生成首部信息（即 "Context-type: text/html\n\n"）。

⊖ 属性前面不要求出现前导短横线。

3. start_html 函数表明开始 HTML 文档，包括了页面标题和背景颜色（灰绿色）。
4. b 函数负责生成 <b> 标签，以便创建粗体字效果；而 font 函数则能让字体增大两号，并将字体颜色设置为深绿色。
5. p 函数负责生成一个段落标签，它不带任何参数，只是一个起始标签。
6. 这个 p 函数则以一个字符串为参数，并将它打印为单独的一段。
7. p 函数的第一个参数将让整个段落位于页面中央。在浏览器中，该段文本将显示为一行内容：red green yellow。
8. 这个段落标签负责在单独的一行靠左显示每个单词。它含有一个起始标签，并对列表中的每一个参数都设定了靠左排列的属性。其参数是一个指向匿名数组的引用。
9. 本行代码将创建结尾的 </body> 和 </html> 标签。

示例 16.26

```
 <!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
  <head>
     <title>Testing arguments</title>
     <meta http-equiv="Content-Type" content="text/html;charset=iso-
   8859-1" />
  </head>
  <body bgcolor="#99FF66">
  <b /><font color="#006600" size="+2" /><p />
  <p>This is a string</p>
  <p align="center">red green yellow</p>
  <p align="left">red</p> <p align="left">green</p> <p
align="left">yellow</p>
  </body>
  </html>
```

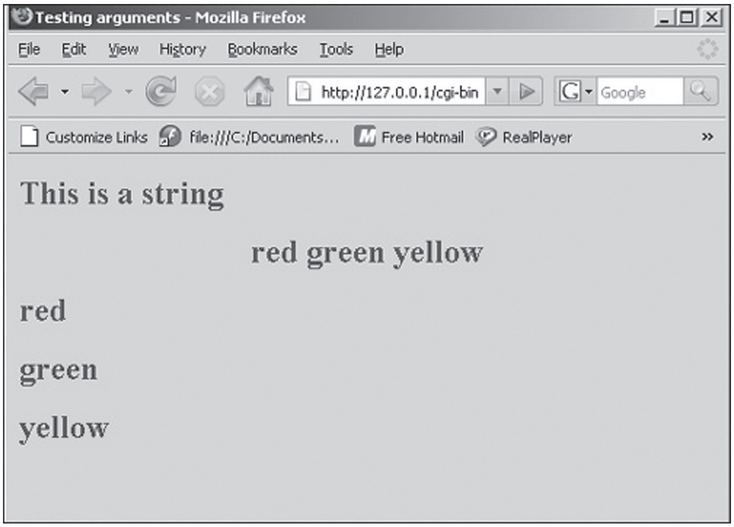


图 16-24　CGI.pm 函数产生的输出内容

为了避免冲突情况和警告信息（-w 开关），读者可以将所有参数放到引号中去。下表 16-11 列出了 GCI.pm 模块提供的一些最为常用的 HTML 方法。

表 16-11　HTML 方法

方　法	用　途	属　性
a{}	锚标签，<*A*> 示例： `print a({-href://google.com/'}, "Go Search"};`	*-href, -name, -onClick, -onMouseOver, -target*
b{}	粗体文本 ,<*B*> 示例： `print b("This text is bold");`	
basefont() (:html3)	设置基本字体的大小，<*FONT*> 示例： `print basefont({-size=7});`	*-size (size 1-7)*
big(:netscape group)	增加文本字体的大小，<*BIG*> 示例： `print "Large, big("and larger");`	
b*r()*	创建行中断，<*BR*> 示例： `print "Break line here", br, "New line starts here.";`	
caption(:html3)	在表上方插入标题，<*CAPTION*> `print table(caption(b(Table Caption)), Tr( continue table here...)`	*-align, -valign*
center()(:netscape group)	文本居中 ,<*CENTER*>	似乎不起作用，请使用 <*CENTER*> 标签
cite()	以适当的斜体字创建文本，<*CITE*>	
checkbox()	创建单独命名的复选框和标记	*-check, -selected, -on, -label, -name, -onClick, -override, -force, -value*
checkbox_group()	创建由单个名称链接的一组复选框	*-columns, -cols, -colheaders, -default, -defaults, -labels, -linebreak, -name, -nolabels, -onClick, - override, -force, -rows, -rowheaders, -value, -values*
code()	以等宽字体创建文本，<*CODE*>	
dd()	定义列表的定义项，<*DD*>	
defaults()	为首次提交表单操作创建提交按钮；同时清除旧的参数列表	
dl()	创建定义列表，<*DL*>；详见 *dd()*	*-compact*
dt()	定义列表的词汇部分，<*DT*>	
em()	着重（斜体）的文本	
end_form(),endform()	结束表单，<*/FORM*>	
end_html()	结束 HTML 文档，<*/BODY*><*/HTML*>	
font()(:netscape group)	更改字体	*-color, -face, -size*

（续）

方 法	用 途	属 性
frame()(:netscape group)	定义一个框架	*-marginheight, - marginwidth, -name, -noresize, -scrolling, -src*
frameset()(:netscape group)	创建一个框架，*<FRAMESET>*	*-cols, -rows*
h1() ... h6()	创建标题级别 1-6 *<H1>,<H2> ... <H6>*	
hidden()	创建一个隐藏、不可见的且用户不能编辑的文本字段	
hr()	创建水平标尺，*<HR>*	*-align, -noshade, -size, -width*
i()	创建斜体文本	
img()	创建行内图像，*<IMG>*	*-align, -alt, -border, -height, -width, -hspace, -ismap, -src, -lowsrc, -vrspace, -usemap*
image_button()	生成一个行内图像，并作为提交按钮图像	*-align, -alt, -height, -name, -src, -width*
kbd()	使用键盘样式创建文本	
li()	创建有序或无序列表的列表项	*-type, -value*
ol()	开始一个有序列表	*-conpact, -start, -type*
P()	创建段，*<P>*	*-align, -class*
password_field()	创建口令字段；输入的文本将显示为星号	
popup_menu()	创建弹出菜单，*<SELECT><OPTION>*	*-default, -labels, -name, -onBlur, -onChange, -onFocus, -override, -force, -value, -values*
pre()	创建预先格式化的打字机文本，用于保持行中断，*<PRE>*	
radio_group()	创建一组链接到同一名字上的单选按钮	*-columns, -cols, -colheaders, -default, -labels, -linebreak, -name, -nolabels, -onClick, -override, -force, -rows, -rowheaders, -value, -values*
reset()	创建表单的 Reset 按钮	
scrolling_list()	控制滚动列表框的表单元素	*-default, -defaults, -labels, -multiple, -name, -onBlur, -onChange, -onFocus, -override, -force, -size, -value, -values*
Select()	创建选择标签；注意必须大写“S”，以免和 Perl 内建的 select 函数冲突，*<SELECT>*	
small()(:netscape group)	减小文本大小	
start_form(), startform()	开始 HTML 表单，*<FORM>*	
start_multiple_form()	与 start_form() 一样，但它用于上传文件	
strong()	粗体文本	
submit()	创建表单的提交按钮	*-name, -onClick, -value, -label*
sup()(:netscape group)	上标文本	

（续）

方 法	用 途	属 性
table()(:html3 group)	创建表	*-align, -bgcolor, -border, -bordercolor, -bordercolor-dark, -bordercolor-light, -cellpadding, -hspace, -vspace, -width*
td()(:html3 group)	创建表单元，*<TD>*	*-align, -bgcolor, -bordercolor, -bordercolor-light, -bordercolor dark, -colspan, -nowrap, -rowspan, -valign, -width*
textarea()	创建多行文本框	*-cols, -columns, -name, -onChange, -onFocus, onBlur, -onSelect, -override, -force, -value, -default, -wrap*
textfield()	生成单行文本字段	*-maxLength, -name, -onChange, -onFocus, -onBlur, -onSelect, -override, -force, -size, -value, -default*
th()(:html3 group)	创建表的标题，*<TH>*	
Tr()(:html3 group)	定义表行；注意其大写字母“T”，以免与 Perl 的内建函数冲突，*<TR>*	*-align, bgcolor, -bordercolor, -bordercolordark, -bordercolorlight, -valign*
tt()	Typewriter 字体	
ul()	开始一个无序列表	

16.7.6 CGI.pm 如何处理表单

CGI.pm 使得用户能够在一个脚本内创建并处理表单。（读者也可创建单独的一个 HTML 表单，同时只使用 CGI.pm 的表单处理部分。不过这里的示例只介绍了如何把整个处理流程集成到一个脚本中去。）该模块提供了许多方法和函数，能够处理 Web 页面创建工作的每一个方面，从新建一个 HTML 文档，一直到创建表单和用于解析和解码其输入数据的输入设备。用户将会在同一个页面中实现表单的生成和其输入的处理。这些脚本都是自处理（self-processing）的，也就是说当用户点击 Submit 按钮时，会激活 ACTION 属性中之前负责打印表单内容的那个脚本。该脚本只需简单地进行一些判断，就能保证该表单已经提交完毕了。读者将在后面看到这些都是如何做到的。

start_html 方法。在创建 HTTP 首部后，大多数 CGI 脚本都会直接开始 HTML 文档。start_html() 方法负责创建页面顶部内容，并通过各种选项规定页面的外观和行为，譬如其背景色、标题、DTD、作者等。它会创建一个首部行（“Contect-type”），此外还会为该文档开始一个 <body> 标签。

start_form 方法。表单是通过 start_form 方法及其参数而创建的。其中每种域类型（field type）也都是一种由模块提供的方法，譬如 textfield、popup_menu、submit，等等。这些方法都列举在表 6.11 中。本章后面将详细介绍其中的每一种方法。

submit 方法。当然，为了处理一个表单，必须首先提交（submit）它。submit 方法能够提供表单上的 Submit 按钮。

param 方法。这是 CGI.pm 中非常重要的一个组件。当用户填写完表单后，CGI.pm 便会从表单中获取输入，进行解码，并将它们保存到键 / 值对中。这些工作都是对用户透明的。其键名和值都可以通过 param() 函数予以检索。在调用 param() 时，如果返回值为 null，就说明用户还没有填好表单。如果 param() 函数返回 true（非 null），则该表单肯定已经填写完毕，此时 param() 函数还可

用于检索表单信息。如果用户想获得某个值，则可使用 param() 函数根据其名称来检索；即根据表单中输入设备的名称。

下面这个示例展示了 CGI 程序的两个组成部分：HTML 表单，以及如何通过 param() 函数得到信息。若想了解其他用于处理参数的函数列表，请参阅表 16-13。

示例 16.27

```
    #!c:/ActivePerl/bin/perl.exe
1   use CGI qw(:standard);
2   print header;
3   print start_html(-title=>'Using the Function-Oriented Syntax',
                  -BGCOLOR=>'yellow');
4   print img({-src=>'/greenballoon.jpg', -align=>LEFT}),
     h1("Let's Hear From You!"),
     h2("I'm interested."),
5    start_form,
6        "What's your name? ", textfield('name'),
         p,
         "What's your occupation? ", textfield('job'),
         p,
7        "Select a vacation spot. ", popup_menu(
            -name=>'place',
            -values=>['Hawaii','Europe','Mexico', 'Japan' ],
                                              ),
         p,
8        "Do you want a green balloon? ", br,
          checkbox(-name=>'choice',-label=>'If yes, check this box'),
         p,
9        submit("Press here"),
10       end_form;
         print hr;
11       if ( param() ){  # If the form has been filled out,
                          # there are parameters
12            print "Your name is ", em(param('name')),
              p,
               "Your occupation is ", em(param('job')),
              p,
               "Your vacation spot is ", em(param('place')),
              p;
13            if( param('choice') eq "on"){
                  print "You will receive a green balloon shortly!"
              }
              else{
                  print "Green may not be the best color for you.";
             }
         hr;
       }
```

解释

1. use 指示符表明正在载入 CGI.pm 模块，并导入函数调用的 :standard 集合，该集合使用了版本 1.21 及其后版本库的新语法。该语法允许调用方法，而不必显式地使用 new 构造函数方法创建对象；即由模块创建对象。Lincoln Stein 的 Official Guide to programming with CGI.pm 中提供了快捷方式的完整列表。

2. 首部方法 header 返回 Content-type: header。读者可以另外选定自己的 MIME 类型，如果不选的话，则默认为 text/html。
3. 返回 HTML 首部，并打开 <*BODY*> 标签。其参数是可选的，其格式包括 -title、-author、base 等。用户还可以将额外的参数（如 Netscape 提供的非官方的 BGCOLOR 属性）附加到 <*BODY*> 标签上；例如，BGCOLOR=>yellow。
4. img 方法用于加载图像。GIF 图像存储在文档根目录的 Images 子目录中。它在文本中是左对齐的。请注意，print 函数直到第 11 行才结束。所有的 CGI 函数都通过以逗号隔开的参数列表传递给 print 函数。
5. 本行会生成一级标题标签。这是产生 <*H1*> 标签的快捷方式。
6. 该方法开始表单。表单默认的 ACTION 属性负责指定该脚本的 URL，而 METHOD 属性则指定了 POST 方法。
7. textfield 方法负责创建文本字段框。其第一个参数是字段的 NAME，第二个参数表示字段的 VALUE，后者是可选的。这里给 NAME 赋值为 name，而把 VALUE 赋值为 “ ”。
8. p 是段标记 <P> 的快捷方式。
9. popup_menu 方法负责创建菜单。其要求的第一个参数是菜单名称（-name）。第二个参数 -value 则是表示菜单项的数组，它可以是命名的，也可以是匿名的。
10. submit 方法能创建 Submit 按钮。
11. 该行表示结束表单。
12. 如果 param 方法返回非空值，则会打印每个关联到参数上的值。
13. param 方法返回与 name 关联的值；也就是用户针对该参数输入的内容。

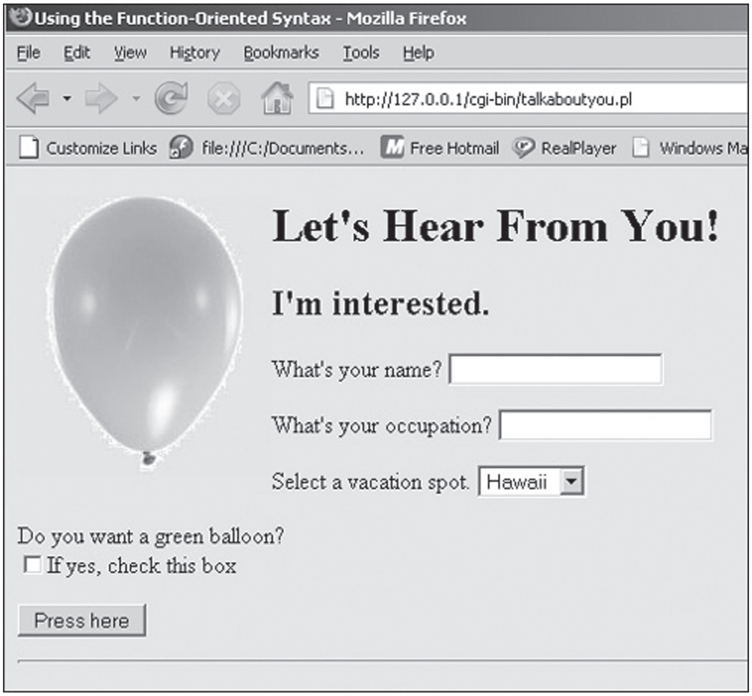


图 16-25 示例 16.27 中填充表单前的 1-11 行输出内容

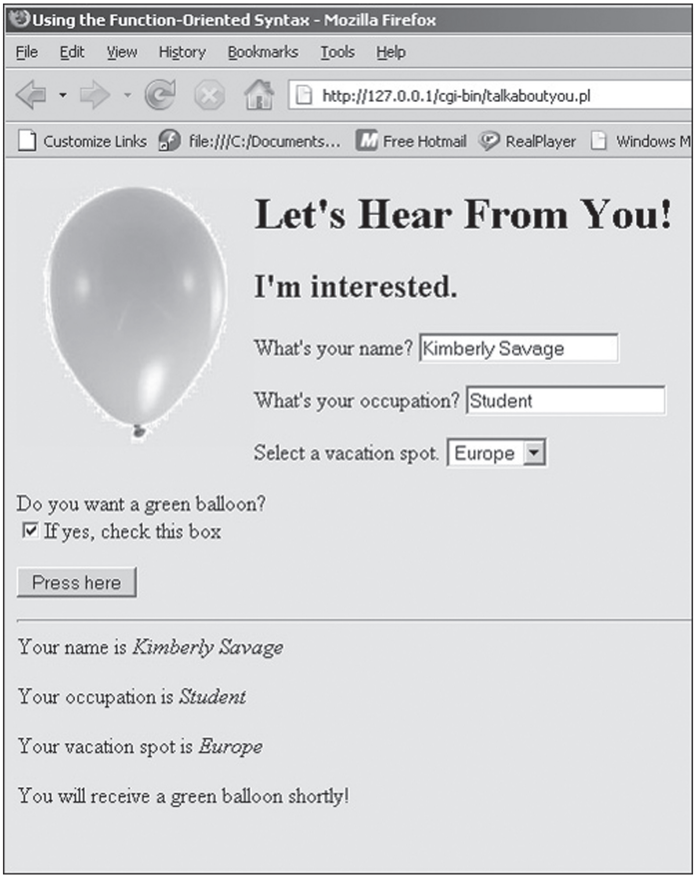


图 16-26　完整的表单和 CGI.pm 处理结果

在命令行中检查表单。 读者如果想要查看 CGI.pm 表单生成的 HTML 标签，可以在命令行中直接运行脚本。不过这样做读者可能只会看到如下错误消息：

```
(Offline mode: enter name=value pairs on standard input)
```

为了处理这种错误，读者可以在键入键 / 值对后按下 <Ctrl>-d（UNIX）或 <Ctrl>-z（Windows），也可传递一个空的参数列表。当参数列表为空时，读者便可在输出中看到没有赋值而生成的 HTML 标签。详见示例 16.28。

示例 16.28

```
(At the Command Line)
    $ perl talkaboutyou.pl
    (Offline mode: enter name=value pairs on standard input)
    name=Dan
    job=Father
    place=Hawaii
    <Now press Ctrl-d or Ctrl-z>

(Output)
```

```
    Content-Type: text/html

    <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
    <HTML><HEAD><TITLE>Using the Function Oriented Syntax</TITLE>
    </HEAD><BODY BGCOLOR="yellow">
    <H1>Let's Hear From You!</H1>
    <H2>I'm internested.</H2>
    <FORM METHOD="POST"  ENCTYPE="application/x-www-form-urlencoded">
        What's your name? <INPUT TYPE="text" NAME="name"
        VALUE="Dan"><P>What's your occupation? <INPUT TYPE="text"
        NAME="job" VALUE="Father"><P>Select a vacation spot.
        <SELECT NAME="place">
    <OPTION SELECTED VALUE="Hawaii">Hawaii
    <OPTION  VALUE="Europe">Europe
    <OPTION  VALUE="Mexico">Mexico
    <OPTION  VALUE="Japan">Japan
    </SELECT>
    <P><INPUT TYPE="submit" NAME=".submit"></FORM>
        <HR>Your name is <EM>Dan</EM><P>Your occupation is <EM>Father
        </EM><P>Your vacation spot is <EM>Hawaii</EM><HR>

(At the Command Line)
2   $ perl talkaboutyou.pl < /dev/null  or perl talkaboutyou.pl ' '
    Content-Type: text/html
(Output)
    <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
    <HTML><HEAD><TITLE>Using the Function Oriented Syntax</TITLE>
    </HEAD><BODY BGCOLOR="yellow">
    <H1>Let's Hear From You!</H1>
    <H2>I'm interested.</H2><FORM METHOD="POST"
        ENCTYPE="application/x-wwwform-urlencoded">
        What's your name? <INPUT TYPE="text" NAME="name"
        VALUE=""><P>What's your occupation? <INPUT TYPE="text"
        NAME="job" VALUE=""><P>Select a vacation spot.
        <SELECT NAME="place">
    <OPTION  VALUE="Hawaii">Hawaii
    <OPTION  VALUE="Europe">Europe
    <OPTION  VALUE="Mexico">Mexico
    <OPTION  VALUE="Japan">Japan
    </SELECT>
    <P><INPUT TYPE="submit" NAME=".submit"></FORM><HR>Your name is
        <EM><P>Your occupation is <EM><P>Your vacation spot is <EM><HR>
```

解释

1. 当在离线模式下运行时，可以将键 / 值对作为标准输入。这里读者需要先检查表单，然后才能获得正确的键，并自行提供相应的值。在本示例中，用户提供的是 name=Dan,job=Father, place=Hawaii。当用户按下 <Ctrl>-d（UNIX）或 <Ctrl>-z（Windows）之后，将由 CGI.pm 负责处理输入。
2. 通过传递一个空参数列表，便能看到其 HTML 输出是没有赋值的。在 UNIX 系统中，/dev/null 是 UNIX 的比特存储器（黑洞），从该目录读取的效果等价于读取空文件。此外也可提供一组空引号作为参数，其效果也是一样的。

16.7.7 CGI.pm 表单元素

表 16-12 CGI 方法

方 法	示 例	用 途
append	*$query->append(-name=>'value');*	将值附加到参数上
checkbox	*$query–>checkbox(-name=>'checkbox_name', -checked=>'checked', -value=>'on', -label=>'clickme');*	创建单独的复选框
checkbox_group	*$query–>checkbox_group(-name=> 'group_name', -values=>[list], -default=>[sublist], -linebreak=>'true', -labels=>\%hash);*	创建一组复选框
cookie	*$query–>cookie(-name=>'sessionID', -value=>'whatever', -expires=>'+3h', -path=>'/', -domain=>'ucsc.edu', -secure=>1);*	创建一个 Netscape cookie
defaults	*$query->defaults;*	创建一个能把表单重置为默认值的按钮
delete	*$query->delete('param');*	删除参数
delete_all	*$query->delete*	删除所有参数；清空 $query，对象
endform	*$query->endform;*	结束 <FORM> 标签
header	*$query->header(-cookie=>'cookiename');*	把一个 cookie 放入 HTTP 首部
hidden	*$query->hidden(-name=>'hidden', -default=>[list]);*	创建隐藏的文本字段
image_button	*$query-> image_button (-name=>'button', -src=>'/source/URL', -align=>'MIDDLE');*	创建可点击的图像按钮
import_names	*$query-> import_names('namespace');*	将变量导入命名空间
keywords	*@keywords = $query->keywords;*	从 Isindex 获取解释后的关键字，并返回一个数组
new	*$query->new CGI;*	解析输入，并将其放入 GET 和 POST 方法的 $query 对象
	$query->new CGI(INPUTFILE);	从前面打开的文件句柄中读取表单内容
param	*@params=$query->param(-name=>'name', -value=>'value');*	返回传入脚本的参数名数组
	$value = $query->('arg');	返回传递参数中对应于 @value = $query -> ('arg'); 的值
password_field	*$query–>password_field(-name=>'secret' -value=>'start', -size=>60, -maxlength=>80);*	创建口令字段
popup_menu	*$query–>popup_menu(-name=>'menu' -values=>@items, -defaults=>'name', -labels=>\%hash);*	创建弹出菜单
radio_group	*$query–>radio_group(-name=>'group_name', -values=>[list], -default=>'name', -linebreak=>'true', -labels=>\%hash);*	创建一组单选按钮
reset	*$query->reset;*	创建 Reset 按钮，负责清除表单框，并恢复为前一个值
save	*$query->save(FILEHANDLE);*	将表单状态保存到文件

（续）

方 法	示 例	用 途
scrolling_list	*$query–>scrolling_list(-name=>'listname', -values=>[list], -default=> [sublist], -multiple=>'true', -labels=>\%hash);*	创建一个滚动列表
startform	*$query->startform(-method=> -action=>, -encoding);*	返回一个 *<FORM>* 标签，使用可选的方法、操作和编码
submit	*$query->submit(-name=>'button', -value=>'value');*	为表单创建 Submit 按钮
textarea		与文本字段（textfield）类似，但它可以含有多行文本输入框
textfield	*$query-> textfield (-name=>'field', -defualt=>'start', -size=>50, -maxlength=>90);*	创建一个文本字段框

表 16-13　CGI 参数方法

方 法	用 途	示 例
delete(),Delete()	从参数列表中删除一个命名参数。在使用面向函数方式的 CGI.pm 时，必须使用 Delete 方法	*$obj->delete('Joe');* *$obj->delete(-name=>'Joe');* *Delete('Joe');* *Delete(-name=>'Joe');*
delete_all(),Delete_all()	删除所有 CGI 参数	*$obj->delete_all();Delete_all();*
import_names()	将所有 CGI 参数导入到指定命名空间中	
param()	以键 / 值对的形式从填好的表单中检索参数。能返回一个列表或标量	*print $obj->param();* *@list=$obj->param();* *print param('Joe');* *$name=$obj->param(-name=>'Joe');*

16.7.8　生成表单输入字段的方法

下面的几个示例将使用面向对象的样式，读者也可很容易地通过删除所有对象引用来把它转换为面向函数的样式。子例程 print_form 能在浏览器窗口中显示表单；而 do_work 子例程将在 param 方法返回真值时负责生成输出内容，这就意味着完成表单的填写和处理。

textfield() 方法。 textfiled 方法能创建一个文本字段。文本字段组件允许用户在一个矩形框中输入单行文本内容。用户可使用 -size 参数规定框的大小，其单位是字符宽度；也可使用 -length（正整数）来设置可以输入的字符数上限。如果没有规定 -maxlength，则默认可以输入任意数量的字符。当使用 -value 时，便可规定该字段的默认值，即第一次显示时出现在框内的文本。

格式

```
print $obj->textfield('name_of_textfield');
print $obj->textfield(
    -name=>'name_of_textfield',
    -value=>'default starting text',
    -size=>'60',          # Width in characters
    -maxlength=>'90');    # Upper width limit
```

示例 16.29

```
#!/usr/bin/perl
1   use CGI;
2   $query = new CGI;
    # Create a CGI object
3   print $query->header;
4   print $query->start_html("Forms and Text Fields");
5   print  $query->h2("Example: The textfield method");

6   &print_form($query);
7   &do_work($query) if ($query->param);
    print $query->end_html;

8   sub print_form{
9       my($query) = @_;
10      print $query->startform;
        print "What is your name? ";
11      print $query->textfield('name');  # A simple text field
        print  $query->br();
12      print "What is your occupation? ";
13      print $query->textfield(-name=>'occupation',  # Giving values
                                -default=>'Retired',  # to the
                                -size=>60,            # text field
                                -maxlength=>120,
                               );
        print $query->br();
14      print $query->submit('action', 'Enter ');
15      print $query->reset();
16      print $query->endform;
        print $query->hr();
    }
17  sub do_work{
        my ($query) = @_;
        my (@values, $key);
        print $query->("<H2>Here are the settings</H2>");
18      foreach $key ($query->param) {
            print "$key: \n";
19          @values=$query->param($key);
            print join(", ",@values), "<BR>";
        }
    }
```

解释

1. 加载 CGI.pm 模块。它是一种面向对象的模块。
2. 调用 CGI 构造函数方法 new，返回指向 CGI 对象的引用。
3. 打印 HTML 首部信息；例如，Content-type: text/html。
4. start_html 方法负责产生 HTML 标签，从而开始 HTML 文档。此外还生成标题 Forms and Textfields，以及 body 标签。
5. h2 方法负责生成 <H2> 标签，即二级标题。
6. 调用用户定义的 print_form 函数，并通过参数传递指向 CGI 对象的引用。
7. 以指向 CGI 对象的引用作为参数调用 do_work 函数。这是用户自定义的函数，只有当 param 函数返回真值时才予以调用。而 param 函数只有在填充好表单后才会返回真。
8. 定义 print_form 函数。

9. 第一个参数是一个指向 CGI 对象的引用。
10. startform 方法产生 HTML 的 <FORM> 标签。
11. textfield 方法使用参数 name 来生成文本框，并把填入文本框的内容赋值给 name。
12. 要求用户在文本框中提供输入。
13. 向 textfield 方法传递散列键 / 值对参数，以便进一步定义文本字段。框中将显示其默认值。参见图 16-27 中显示的该示例输出内容。
14. submit 方法能创建提交按钮，按钮上的文本内容是 Enter。
15. reset 方法负责创建重置按钮，该按钮上的文本内容是 Reset。
16. endform 方法负责创建 HTML 的 </FORM> 标签。
17. 这是用户自定义的 do_work 函数，当用户填好表单并按下提交（Enter）按钮后，程序就会调用该方法。它使用 param 函数来处理表单提交的内容。
18. param 函数能返回一个键以及关联到这个键的值列表。这里的键是输入表单的参数名，值则是用户或表单赋予它的内容。例如，用户会把键 occupation 的对应值填入为 jack of all trades；而在表单处理之前，提交按钮中 action 键的相应值为 Enter。

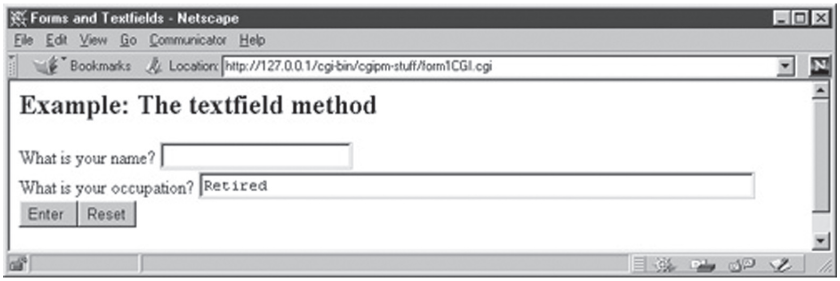


图 16-27　示例 16.29，文本字段表单的输出

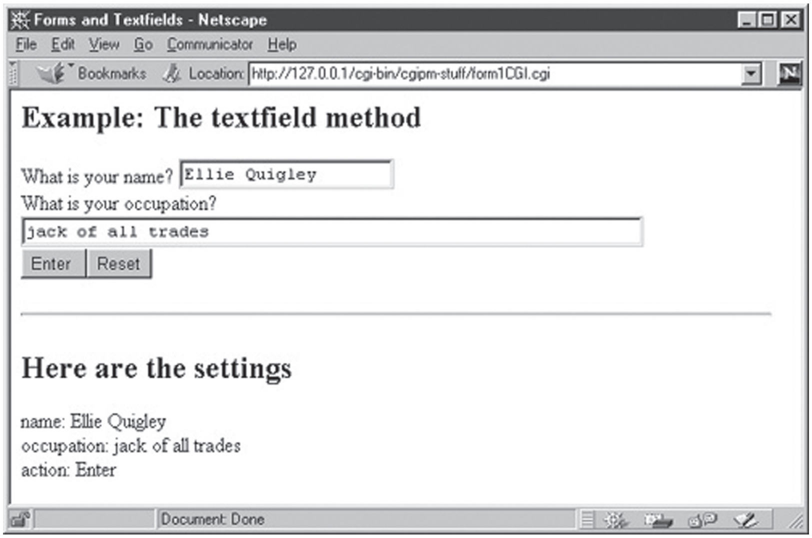


图 16-28　示例 16.29，填充并处理表单后的输出

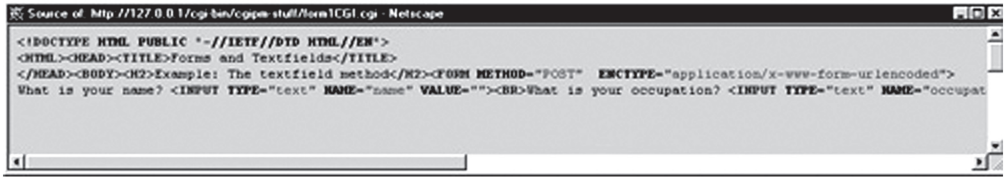


图 16-29　CGI.pm 生成的 HTML 源代码

checkbox() 方法。　checkbox() 方法用于创建以 yes 或 no（布尔值）响应的简单复选框。复选框具有 NAME 和 VALUE 属性，其中 -name 负责给 CGI 参数命名，而 -value 则含有单个项或者一个指向可选列表项的引用。如果指定 -checked 为 1，则复选框在开始时就是选中的。如果给 -label 赋了值，则在复选框下面打印其内容；否则就打印复选框的 -name 值。如果没有选定该复选框，它就含有空参数。

checkbox_group() 方法负责创建一组复选框，并将它们链接到同一个名字上。各个选项之间并不相互排斥；即，用户可以同时选择其中一项或多项。如果把 -linebreak 赋值为非零数字的话，就表明垂直排列其选项。一般情况下则把选项显示为水平的一行（详见示例 16.30）。

格式

```
print $obj->checkbox(-name=>'name_of_checkbox',
                     -checked=>1,
                     -value=>'ON'
                     -label=>'Click on me'
                    );

%labels = ('choice1'=>'red',
           'choice2'=>'blue',
           'choice3'=>'yellow',
          );
print $obj->checkbox_group(-name=>'name_of_checkbox',
                           -values=>[ 'choice1', 'choice2',
                                      'choice3', 'green',...],
                           -default=>[ 'choice1', 'green' ],
                           -linebreak => 1,
                           -labels=>\%labels
                          );
```

示例 16.30

```
   #!/usr/bin/perl
   use CGI;
   $query = new CGI;
   print $query->header;
   print $query->start_html("The Object Oriented CGI and Forms");
   print "<H2>Example using Forms with Checkboxes</H2>\n";

   &print_formstuff($query);
   &do_work($query) if ($query->param);
   print $query->end_html;
   sub print_formstuff{
       my($query) = @_;
1      print $query->startform;

       print "What is your name? ";
       print $query->textfield('name');    # A simple text field
```

```
        print "<BR>";

        print "Are you married? <BR>";
2       print $query->checkbox(-name=>'Married',
                               -label=>'If not, click me' );
                               # Simple checkbox
        print "<BR><BR>";
        print "What age group(s) do you hang out with? <BR>";
3         print $query->checkbox_group(-name=>'age_group',
                                       -values=>[ '12-18', '19-38',
                                                  '39-58','59-100' ],
                                       -default=>[ '19-38' ],
                                       -linebreak=>'true',
                                      );
4       print $query->submit('action', 'Select');
5       print $query->reset('Clear');
        print $query->endform;
        print "<HR>\n";
    }

6   sub do_work{
        my ($query) = @_;
        my (@values, $key);
        print "<H2>Here are the settings</H2>";
7       foreach $key ($query->param){
            print "$key: \n";
8           @values=$query->param($key);
            print join(", ",@values), "<BR>";
        }
    }
```

解释

1. startform 方法产生 HTML 的 <FORM> 标签。
2. 这是最简单的一种复选框。如果用户尚未婚配，就应当点击该框。该复选框的名字是 Single；程序会在复选框下显示标记 If not, click me。如果把 -checked 赋值为 1 的话，则该复选框在第一次出现时就处于选中状态。
3. 这是复选框组的一个示例。一组相关复选框通过公用的名字 age_group 链接起来。将 -value 赋值为指向选项列表的引用，这些选项将出现于复选框的右侧。如果用到了 -label，它将含有一个由键 / 值对组成的散列，该散列用于规定每一个复选框的标记内容。-default 参数负责决定第一次出现时应当选中哪个复选框。如果将 -linebreak 赋为非零值（真）的话，就表明把选项以垂直列表形式显示出来。复选框的输出内容如图 16-30 所示。
4. 当用户按下标记内容为 Select 的按钮时，处理表单。
5. Reset 按钮只能在用户尚未提交表单时用于清空屏幕内容。如要克服“粘性”，即将复选框设置为原始的默认值，则可将 -override 参数设定为非零值。
6. 在提交表单时调用 do_work 函数。该函数负责处理表单输入数据的读取和解析工作。填写并处理完表单后的输出内容如图 16-31 所示。
7. 打印来自表单（键 / 值对）的每一个参数。

radio_group() 和 popup_menu() 方法。 如需从一组相互排斥的选项中进行选择，可使用单选按钮（radio button）或者弹出菜单（pop-up menu）。单选按钮使得用户能把少量的选项组织在一起，并置于表单中；不过对于大量的选项，还是使用弹出菜单更好。它们都接受由键 / 值对组成的参数。由于其参数值可以由多个选项构成，因此这里得到的是指向数组的引用。-default 参数负责指定菜

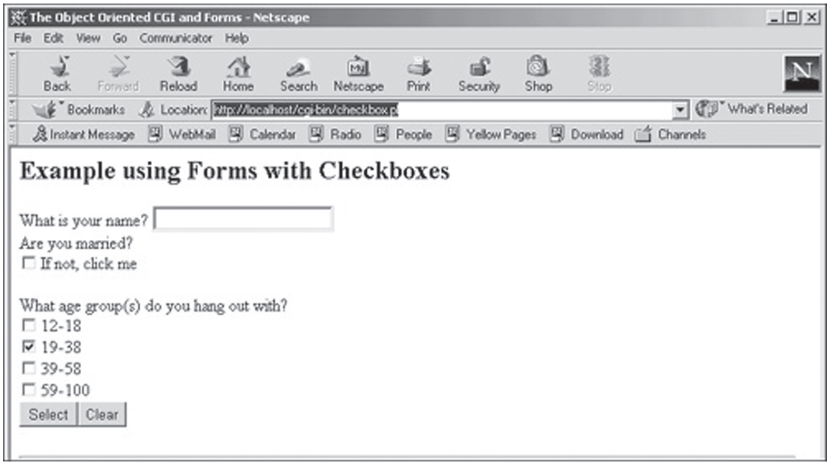


图 16-30　示例 16.30 中复选框的输出

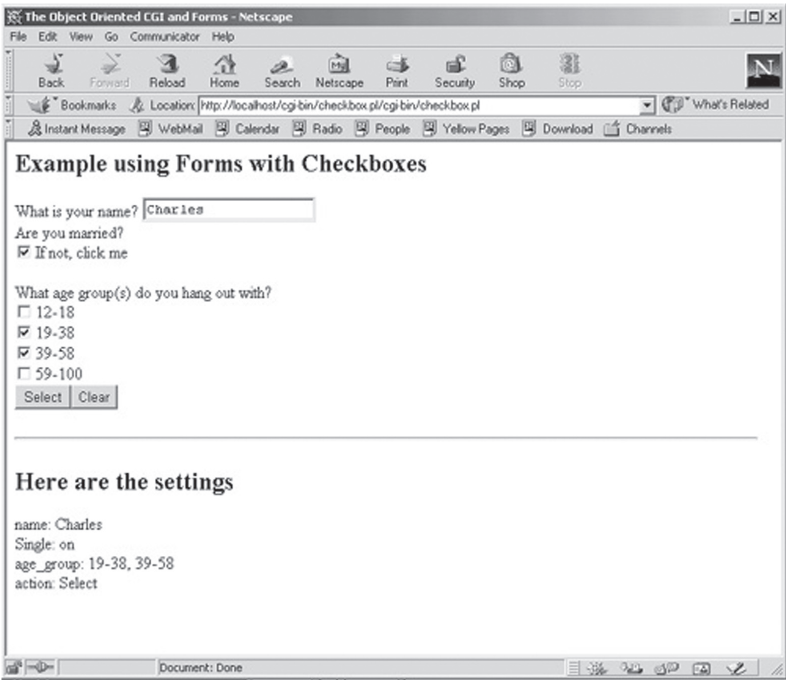


图 16-31　示例 16.30，填写并处理表单后的输出

单第一次出现时所显示的值。如果要在单选组或弹出菜单中使用不同的用户可见标记值，不妨提供可选参数 -label，并将其值返回给脚本。这是一个指向关联数组的指针，负责把菜单值关联到相应的用户可见的标记上。如果没有提供默认值，一开始便会选中弹出菜单的第一项。

格式

```
%labels = ('choice1'=>'red',
           'choice2'=>'blue',
```

```
            'choice3'=>'yellow',
           );
print $obj->radio_group(-name=>'name_of_radio_group',
                        -values=>['choice1','choice2',
                                  'choice3', 'green', ...],
                        -default=>[ 'choice1', 'green' ],
                        -linebreak => 1,
                        -labels=>\%labels
                       );

%labels = ('choice1'=>'red',
           'choice2'=>'blue',
           'choice3'=>'yellow',
          );
print $obj->popup_menu(-name=>'name_of_popup_menu',
                       -values=>['choice1','choice2','choice3',
                                 'green',...],
                       -default=>[ 'choice1', 'green' ],
                       -linebreak => 1,
                       -labels=>\%labels
                      );
```

示例 16.31

```
    #!/bin/perl
1   use CGI;
    $query = new CGI;
    print $query->header;
    print $query->start_html("The Object-Oriented CGI and Forms");
    print "<H2>Example using Forms with Radio Buttons</H2>\n";
    &print_formstuff($query);
    &do_work($query) if ($query->param);
    print $query->end_html;

    sub print_formstuff{
    my($query) = @_;
    print $query->startform;
    print "What is your name? ";
    print $query->textfield('name');  # A simple text field
    print "<BR>";
    print "Select your favorite color? <BR>";
2   print $query->radio_group(-name=>'color',
                              -values=>[ 'red', 'green',
                                         'blue','yellow' ],
                              -default=>'green',
                              -linebreak=>'true',
                             );
    print $query->submit('action', 'submit');
    print $query->reset('Clear');
    print $query->endform;
    print "<HR>\n";
    }
    sub do_work{
        my ($query) = @_;
        my (@values, $key);
        print "<H2>Here are the settings</H2>";
3       foreach $key ($query->param){
            print "$key: \n";
4           @values=$query->param($key);
            print join(", ",@values), "<BR>";
        }
    }
```

解释

1. 加载 CGI.pm 模块。

2. 以参数调用 radio_gourp 方法，在表单中创建一个单选按钮组（radio g’roup）。每个单选按钮的值都出现在按钮右侧。在第一次显示时，表单默认选定的按钮是 green。-linebreak 参数将在屏幕上以垂直方式放置按钮，而不是水平方式。用户只能选择其中一个按钮。单选按钮组的输出内容如图 16-32 所示。
3. 当用户填写好表单并按下提交按钮后，param 方法将返回发送到 CGI 脚本的键和值。
4. 显示键 / 值对。填入并处理完表单后的输出内容如图 16-33 所示。

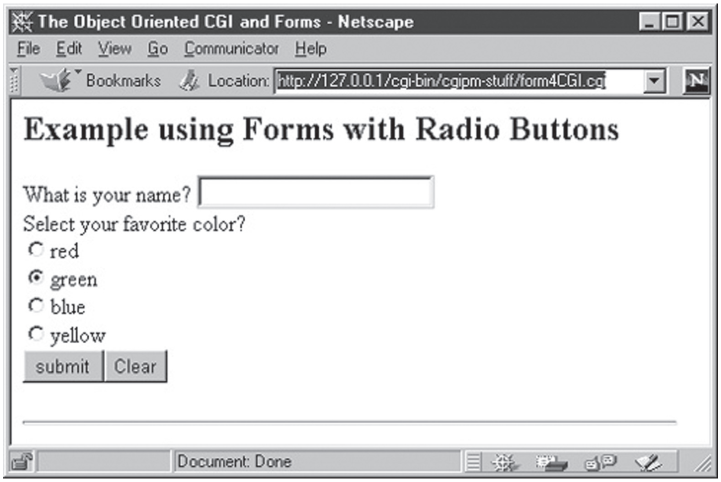


图 16-32　示例 16.31，单选按钮表单的输出内容

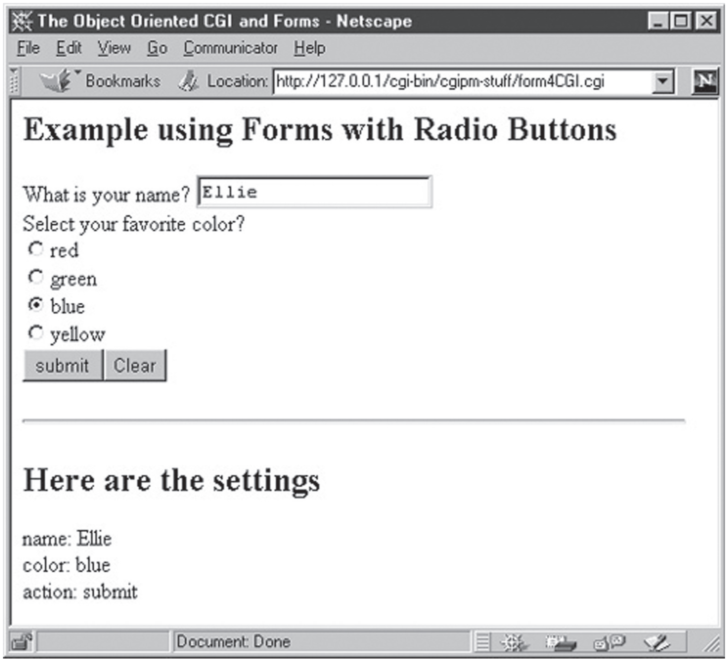


图 16-33　示例 16.31，填写并处理表单后的输出

标记。标记使按钮拥有对用户友好的名字，并在程序中关联到不同的对应值。下面的示例中，标记 stop、go 和 warn 将分别出现在浏览器窗口中单选按钮的旁边。由 param 方法返回的值将分别是 red、green 和 yellow。

示例 16.32

```
  ( The -labels Parameter -- Segment from CGI script)
      print $query->startform;
      print "What is your name? ";
      print $query->textfield('name');     # A simple text field
      print "<BR>";
      print "We're at a cross section. Pick your light.<BR>";
1     print $query->radio_group(-name=>'color',
2                                -values=>[ 'red', 'green', 'yellow' ],
                                 -linebreak=>'true',
3                                -labels=>{red=>'stop',
                                           green=>'go',
                                           yellow=>'warn',
                                          },
4                                -default=>'green',
                                );
      print $query->submit('action', 'submit');
      print $query->reset('Clear');
      print $query->endform;
      }
```

解释

1. 以参数调用 radio_gourp 方法。用户一次只能选择其中一个值。
2. 由 param 方法返回的值分别是 red、green 和 yellow。
3. 标记实际将出现在每个单选按钮的旁边。用户在浏览器窗口中可以看到 stop、go 和 warn，

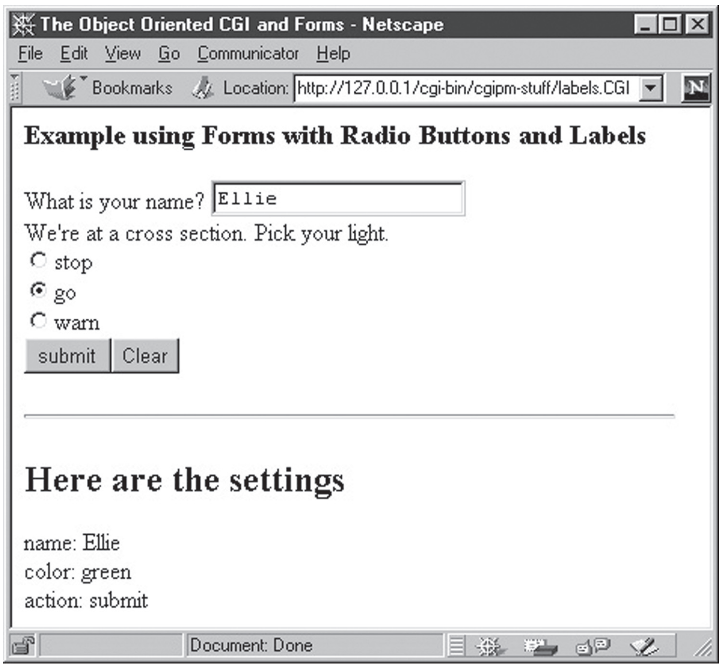


图 16-34　示例 16.32，填写并处理表单后的输出

不过和这些标记相关联的 CGI 参数则是 red、green 和 yellow。例如，如果用户单击 stop 按钮，则传递给 CGI 脚本的键 / 值对将是 color=>red。

4. 默认按钮是标记为 go 的按钮。

popup_menu() 方法。 弹出菜单又称作下拉菜单（drop-down menu）。它是在用户单击文本右侧的滚动条图标时显示出来的选项列表。一次只允许作一项选择。图 16-35 和图 16-36 分别显示了示例 16.33 中弹出菜单的输出内容，以及填入并处理完表单后的输出情况。

示例 16.33

```
#!/usr/bin/perl
use CGI;
$query = new CGI;
print $query->header;
print $query->start_html("The Object-Oriented CGI and Forms");
print "<H2>Example using Forms with Pop-up Menus</H2>\n";
&print_formstuff($query);
&do_work($query) if ($query->param);
print $query->end_html;

sub print_formstuff{
my($query) = @_;
print $query->startform;
print "What is your name? ";
print $query->textfield('name');  # A simple text field
print "<BR>";
print "Select your favorite color? <BR>";
print $query->popup_menu(-name=>'color',
                         -values=>[ 'red', 'green', 'blue',
                                    'yellow' ],
                         -default=>'green',
                         -labels=>'\%labels',
                        );
print $query->submit('action', 'submit');
print $query->reset('Clear');
print $query->endform;
print "<HR>\n";
}

sub do_work{
    my ($query) = @_;
    my (@values, $key);
       print "<H2>Here are the settings</H2>";
       foreach $key ($query->param){
           print "$key: \n";
           @values=$query->param($key);
           print join(", ",@values), "<BR>";
    }
}
```

submit() 和 reset() 方法。 submit() 方法能创建一个按钮，当按下它时，便会将表单输入发送到 CGI 脚本。如果提供了相应参数，用户还可为该按钮提供标记（label），以便在同时使用多个提交按钮时进行区分。

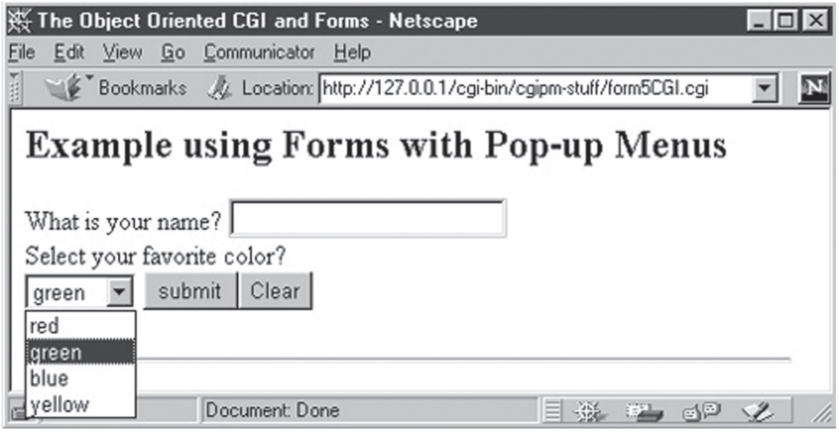


图 16-35 示例 16.33，弹出菜单表单的输出

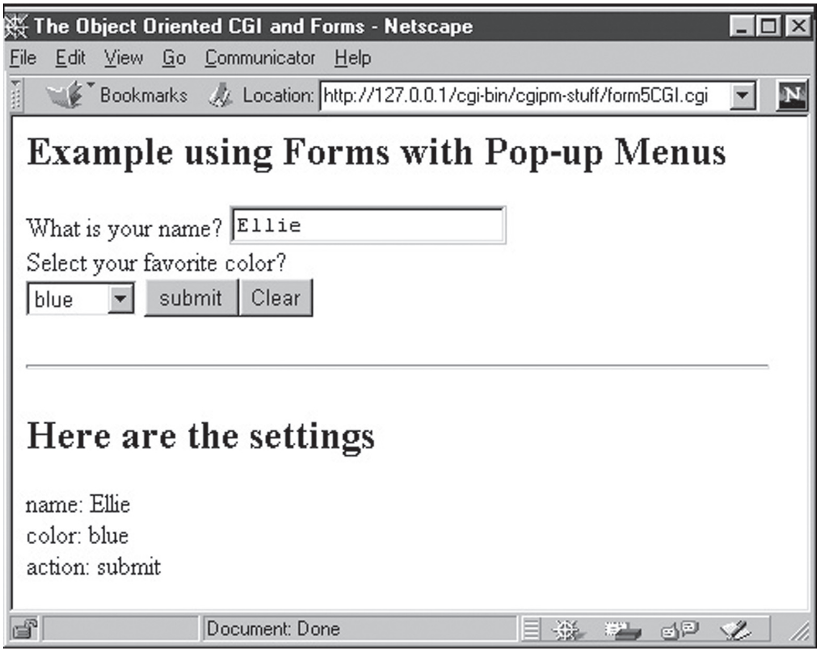


图 16-36 示例 16.33，填写并处理表单后，弹出菜单表单的输出

reset() 方法则用于清除表单中的所有项。它会把表单恢复到上一次加载时的状态，而不是默认的状态（详见下面的“defaults 方法”一节）。

清除字段

override 参数

请注意，如果按下 Reset 按钮或重新启动一个表单，则之前填入的信息还将粘在里面；也就是说，输入框并没有得以清空。这时用户可通过使用非零的 -override 或 -force 参数来强制清空这些项。例如：

```
textfield (-name=>'name' , -override=>1);
```

defaults 方法

defaults() 方法能将表单中所有的项目清空为浏览器窗口第一次显示时的状态；即清空参数列表。如要创建用户可读的按钮，也可调用 defaults 方法；例如：

```
print defaults( -name=>'Clear All Entries');
```

16.7.9 错误处理

当 CGI.pm 脚本中出现错误时，服务器一般会把错误信息发送到配置在服务器根目录下的错误日志文件中。如果程序退出执行，服务器会显示“Document contains no data”或“Server Error”。这些信息都没有什么用处。

carpout 和 fatalsToBrowser 方法。 CGI.pm 提供了一种方法，不但能将错误信息保存到日志文件中，还可以在浏览器窗口中看到致命错误消息的内容。carpout 方法就是为了这个目的而提供的。在默认情况下，程序不会自动导入该方法，因此读者必须显式地导入它，即输入：

```
use CGI::Carp qw (carpout);
```

carpout 函数接受一个参数，即一个指向用户自定义的文件句柄的引用，错误消息都将发送到该文件句柄中。读者应当在 CGI 应用程序顶部的 BEGIN 代码块中调用它，以便能够捕获编译器错误。如要让来自 die、croak 和 confess 的致命错误信息也出现在浏览器窗口的话，还应当导入 fatalsToBrowser 函数。

示例 16.34

```
    #!/usr/bin/perl
1   use CGI;
2   BEGIN{ use CGI::Carp qw(fatalsToBrowser carpout);
3       open(LOG,">>errors.log") ||die "Couldn't open log file\n";
4       carpout(LOG);
    }
    $query = new CGI;
    <Program continues here>
```

解释

1. 加载 CGI.pm 模块。
2. 加载 CGI::Carp 模块。这个 Carp 模块接受两个参数：fatalsToBrowser 和 carpout。第一个参数 fatalsToBrowser 会将 Perl 错误消息发送到浏览器；而 carpout 则能将标准错误信息从屏幕重定向到浏览器和错误日志中。其效果如图 16-37 所示。
3. 打开文件 errors.log，以便执行文件创建 / 追加操作。该日志文件所含错误信息也将发送到浏览器中。
4. carpout 函数会把错误消息发送到 error.log 文件中。下面是该文件中的某一行内容：
   [Thu Feb 8 18:59:04 2001] C:\httpd\CGI-BIN\carpout.pl: Testing error messages from CGI script.

更改默认消息。在默认情况下，软件错误消息后面都会跟着一段面向 Web 管理员的提示信息，该信息附有发生错误的日期和时间，可通过电子邮件的方式通知管理员。如要更改默认消息的内容，则可使用 set_message 方法。不过此前必须将该方法导入到程序的命名空间中。

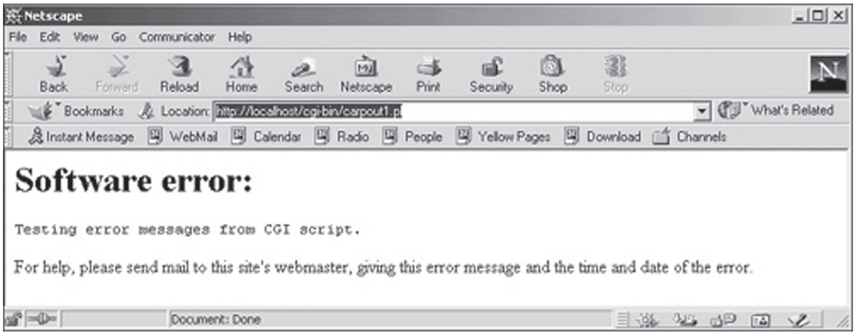


图 16-37 使用 carpout 和 fatalsToBrowser 重定向错误

格式

```
use CGI::Carp qw(fatalsToBrowser set_message);
set_message("Error message!");
set_message(\reference_to_subroutine);
```

示例 16.35

```
1   #!c:/ActivePerl/bin/perl.exe

2   BEGIN{ use CGI::Carp qw(fatalsToBrowser carpout set_message);
3       open(LOG,">>errors.log") ||die "Couldn't open log file\n";
4       carpout(LOG);
5       sub handle_errors {
6           my $msg = shift;
7           print "<h1>Software Error Alert!!</h1>";
            print "<h2>Your program sent this error:<br><I>
               $msg</h2></I>";
        }

7     set_message(\&handle_errors);
    }
8   open(FH, "myfile" ) or
9   die("Can't open \"myfile\": $!\n");
```

解释

1. shbang 行告诉服务器 Perl 解释器位于哪个位置。
2. 在 BEGIN 代码块中载入 CGI.pm 模块，以便立刻捕获任何的编译器错误。参阅 BEGIN 一节。此外还载入了 CGI:Carp 模块。Carp.pm 模块导入了三个函数: fatalsToBrowser、carpout 和 set_message。其中第一个函数 fatalsToBrowser 负责将 Perl 错误发送到浏览器和服务器的错误日志中。carpout 函数用于把标准错误消息重定向到用户自定义的错误日志中。而 set_message 方法则用于定制发送到浏览器的默认错误消息。该函数的参数可以是一个待打印的字符串，也可以是一个用户自定义的函数。
3. 打开文件 errors.log，以便执行文件创建 / 追加操作。该日志文件所含错误信息也将发送到浏览器中。
4. carpout 函数会把错误消息发送到 error.log 文件中。
5. 定义一个名叫 handle_errors 的用户自定义子例程。它将在浏览器窗口中生成一条定制的错误信息。handle_errors 函数的第一个参数是来自 die 或 croak 的错误消息。在本示例中，除非

第 3 行的 die 语句首先执行，否则第 9 行的 die 消息将会赋值为 $msg。该消息同样会发送到日志文件 errors.log 中。

6. 从 @_ 数组中移出错误消息，并发送给浏览器。
7. 调用 set_message 函数，其参数是一个指向用户自定义子例程 handle_errors 的引用，该子例程中含有定制的错误消息。
8. die 函数会让程序退出，并将其错误消息通过 set_message 发送给 handle_errors 子例程。



图 16-38　示例 16.35 中显示的错误消息

示例 16.36

```
(The Apache log file)
[Thu May 24 14:12:00 2007] [error] [client 127.0.0.1] C:/wamp/A
pache2/cgi-bin/errorhandling.pl is not executable; ensure inter
preted scripts have "#!" first line
[Thu May 24 14:12:00 2007] [error] [client 127.0.0.1] (9)Bad fi
le descriptor: don't know how to spawn child process: C:/wamp/A
pache2/cgi-bin/errorhandling.pl
(The errors.log file created in the cgi-bin directory by the carpout
function)
[Thu May 24 14:21:16 2007] errorhandling.pl: Testing error mess
ages from CGI script.
[Thu May 24 14:22:16 2007] errorhandling.pl: Can't open myfile:
 No such file or directory
[Thu May 24 14:22:55 2007] errorhandling.pl: Can't open <em>myf
ile</em>: No such file or directory
[Thu May 24 14:23:12 2007] errorhandling.pl: Can't open "myfile
": No such file or directory
```

16.7.10　HTTP 首部方法

前面的示例将 cookie 赋予了一个 HTTP 首部。表 16-14 列出了其他能够在 HTTP 首部中创建和检索信息的方法。

表 16-14 HTTP 首部方法

HTTP 首部方法	用 途
accept()	列出 MIME 类型或者类型
auth_type()	返回当前会话的认证类型
cookie()	创建和检索 cookie
header()	返回有效的 HTTP 首部和 MIME 类型
https()	返回有关一个会话的 SSL 信息
path_info()	设置和检索附加路径信息
path_translated()	返回附加路径信息
query_string()	返回 URL 编码的查询字符串
raw_cookie()	返回从浏览器中发送的未经处理的 cookie 列表
redirect()	生成 HTTP 首部，含有对浏览器的重定向请求，以便加载该位置的页面
referer()	返回在脚本开始执行前浏览器所显示页面的 URL
remote_addr()	返回远程主机的 IP 地址，有可能是代理服务器地址
remote_host()	返回远程主机的 DNS 名称
remote_ident()	如果激活了身份验证 Daemon，则返回远程用户的登录名
remote_user()	返回用于验证口令的账号名
request_method()	返回 HTTP 方法，GET、POST 或 HEAD
script_name()	返回本脚本相对于服务器根目录的 URL
self_url()	返回 CGI 脚本的 URL：包括协议、主机、端口、路径、附加路径信息和参数列表，可用于再次调用当前的脚本
server_name()	返回 Web 服务器的名称
server_port()	返回当前会话的端口号（一般是 80）
server_software()	返回 Web 服务器的名称和版本
url()	返回当前脚本的 URL，不含附加路径信息和查询字符串
user_agent()	返回浏览器信息
user_name()	如果可以的话，返回远程用户的名称
virtual_host()	返回浏览器正在访问的虚拟主机名

示例 16.37

```
    #!/usr/bin/perl
    use CGI qw(:standard);
1   print header;
2   print start_html(-title=>'Using header Methods'),
    h1("Let's find out about this session!"),
    p,
3   h4 "Your server is called ", server_name(),
```

```
    p,
    "Your server port number is ", server_port(),
    p,
    "This script name is: ", script_name(),
    p,
    "Your browser is ", user_agent(), "and it's out of date!",
    p,
    "The query string looks like this: ", query_string(),
    p,
    "Where am I? Your URL is: \n", url(),
    p,
    "Cookies set: ", raw_cookie();

print end_html;
```

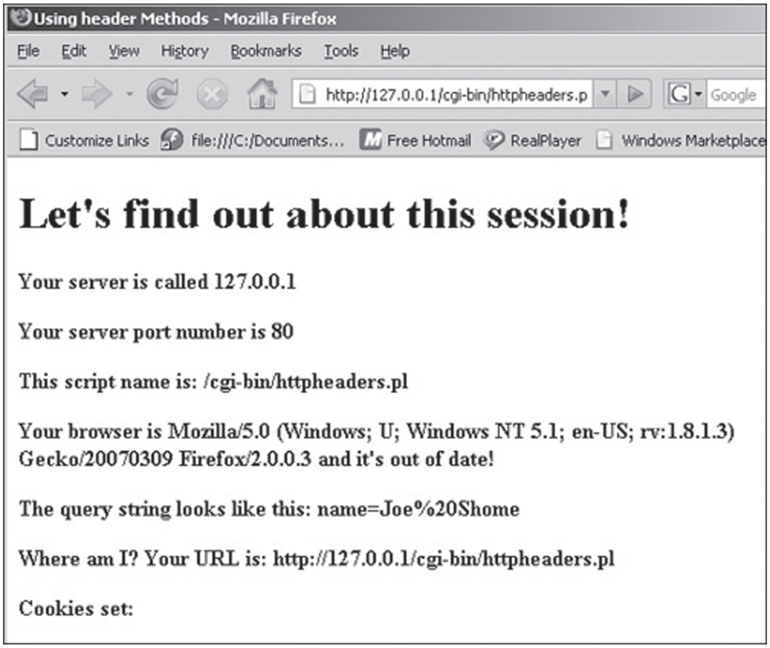


图 16-39　示例 16.37 的输出

练习 16　为 Perls 冲浪

第一部分

1. 环境变量和 CGI

   创建一个 CGI 脚本，向浏览器打印如下内容：

   The name of the server is: <Put the values here.>

   The gateway protocol is:

   The client machine's IP address:

   The client machine's name:

   The document root is:

   The CGI script name is:

（提示：不妨使用 %ENV 散列）

2. 生成 CGI 程序
   a. 编写一个名叫 town_crier 的 CGI 脚本，其中含有 HTML 文本和 Perl 语句。
   b. 该脚本含有两个子例程：&welcome 和 &countem。
   c. welcome 子例程将打印 Welcome Sir Richard！！，并以蓝色字体链接 welcome（注意，Internet Explorer 会忽略 blink）。该子程序还将打印当天日期（使用 ctime 库函数）。
   d. 子例程 countem 位于一个名叫 countem.pl 的文件中。town_crier 脚本将调用 countem，并将自己的名字（town_crier）作为参数传递给该子例程。请记住，脚本的名称保存在变量 $0 中，例如 &countem($0);。该子例程将返回页面访问次数。
   e. 参考图 16-40，考虑如何让脚本在浏览器窗口中显示这些输出。
   f. countem 函数应当设计为：
      - 取一个参数：即调用它的文件名。除非目录中已经有了文件 town_citier.log，创建该文件。然后以读写模式打开该文件。（如果从其他脚本调用 countem 函数，则创建的日志文件名将是该脚本的名字后面跟着 .log 扩展名）。
      - 如果新的日志文件内容为空，countem 将把值 1 写入该文件；否则就从文件中读取一行。该行应当含有一个数字。读取该数字并将其保存到变量中。然后将这个数字递增 1。每次执行 town_citier 时都要调用该函数。
      - 将新的数字发送回文件，并覆盖原有的数字。
      - 关闭日志文件。
      - countem 子例程将数字值返回给调用程序（在本例中，笔者会把该数字放入 HTML 表的单元格中，并将整个字符串发送回到 town_citer。如果读者没有时间，也可不创建这个表，而是直接返回该数字）。
      - 如果是在 UNIX 系统上运行的话，则可以在使用文件之前通过 flock 函数加上一个排他锁，并在处理完毕后移除此锁。

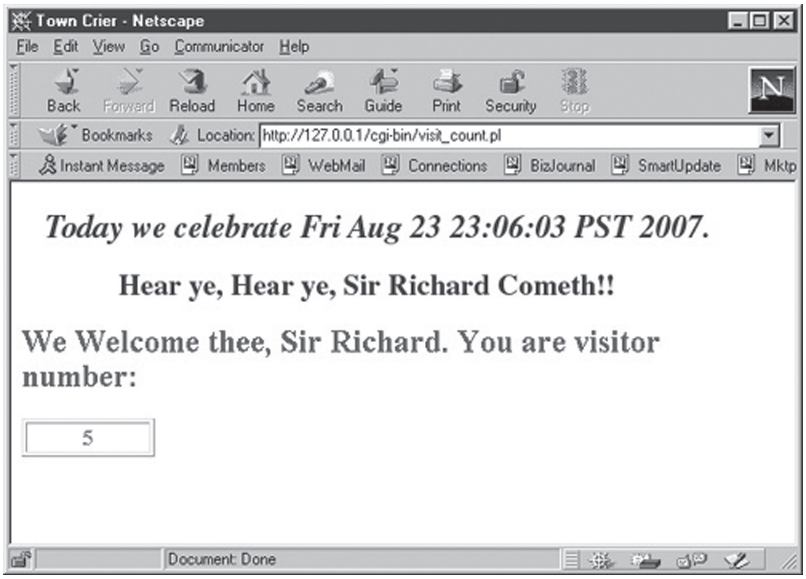


图 16-40　练习 16 中 CGI 程序的输出

第二部分

1. 创建表单：HTML
   a. 创建一个名叫 Stirbucks 的 Web 页面，其中含有一个订购咖啡的表单，类似于图 16-41 所示的订购表单。
   b. 在开始的 <FORM> 标签的 action 属性中，指定一个 URL，以便将服务器指向使用默认的 GET 方法的 CGI 脚本。
   c. 在浏览器中检查文件。
   d. 这段 CGI 脚本将打印环境变量 QUERY_STRING 的值。

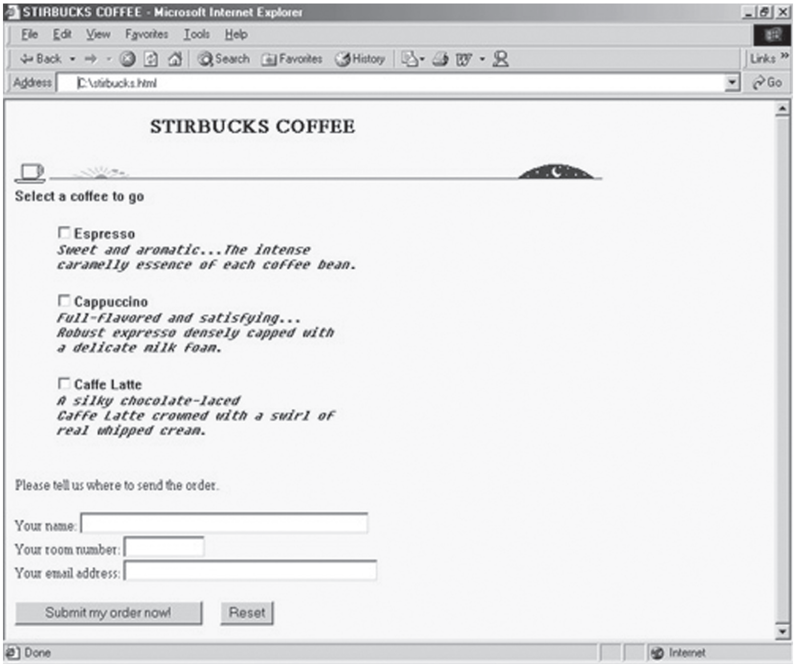


图 16-41　Stirbucks Web 页面上的订购表单

2. 处理表单：CGI
   a. 编写一段 CGI 脚本，向用户发送 HTML 页面，感谢他的订购，并告诉他马上会把他所选择的咖啡递送到他提供的地址。使用 GET 方法。在从表单中得到信息后，请读者编写自己的函数去解析输入内容。
   b. 重新设计表单，使用 POST 方法。程序将判断该使用什么方法，并调用合适的解析函数。
   c. 创建 DBM 文件，其中含有提交的电子邮件地址的列表。当用户提交表单时，将其电子邮件信息列出到 DBM 文件中。请确保其中没有重复现象。请设计一个能完成这项任务的函数。
   d. CGI 脚本将能处理电子邮件。给自己发送电子邮件，确认信息提交成功。请设计另一个负责处理电子邮件的函数。
3. 重写 Stirbucks 程序，将 HTML 表单和 CGI 脚本放入同一个由 CGI.pm 模块创建的 CGI 程序中。请使用面向函数的样式。

第 17 章　当 Perl 遇见 MySQL：完美的连接

17.1　简介

假设用户已经在表单中填好了想要购买的东西，然后提交了该表单。此时，该用户的相关信息就会保存到数据库里一个名叫“customers”的表上。读者可能需要在 Perl 程序中打开这个数据库并直接添加新的订单。此外，还可能需要在一个 Perl 程序中检索该用户以前的订单和产品信息，并对数据进行格式化，以便生成 Web 页面、电子邮件消息、或者发送到电子制表软件（spreadsheet）。有了 Perl 和 Perl DBI 模块，这些都是可以实现的。Perl DBI 模块是一种面向对象的数据库接口，允许用户连接到任意的关系型数据库上，并使用 Perl 方法执行所有必需的操作，譬如打开和关闭数据库，发送用于创建、刷新和删除数据表的 SQL 查询语句，检索和修改记录，管理事务（transaction），以及显示结果。

本章将专门介绍如何在 Perl 中访问 MySQL 数据库，后者是一种非常流行的开源、全功能的关系型数据库。读者将首先学习如何在 MySQL 客户端中发布命令，然后使用 DBI 模块在一段 Perl 脚本中发布相同的命令。最后，本书将把这些内容结合到一起，同时使用 DBI 模块和 CGI.pm 来创建一个动态 Web 页面。

与数据库相关的主题非常繁杂。本章的目标并不是试图教会读者如何正确地设计一个数据库的结构，或者如何更好地组织数据。这些内容又可以占据一本书的篇幅。如果读者完全是一个新手，从来没有和数据库打过交道，也不了解它们的工作原理，那么可以阅读一下 Andy Oppel 撰写的 Databases Demystified，这是一本非常出色的数据库入门书籍[⊖]。本章将简要介绍一下使用 Perl DBI 和 MySQL 时必须了解的一些基本概念和专业术语。

17.2　什么是关系型数据库

迄今为止我们在 Perl 中都是把数据保存到普通文本文件中的，并为此创建了用户自定义的文件句柄。此外，我们还了解了 Perl 的 DBM 机制，它能将数据以简单散列的形式保存到二进制文件里去。不过，当用户需要高效地存储和管理大量数据时，上述文本文件和 DBM 文件就显得力不从心

⊖ Oppel, Andrew J. Databases Demystified. McGraw-Hill/Osbourne, Emeryville, CA, 2004.

了：例如，维护一个商业应用，如医院、研究实验室、银行、学校或 Web 站点。关系型数据库系统遵循某些特定的标准，并提供了多种特性，以便存储大批量的数据。这些数据都受到很好的管理，从而让检索、更新、插入和删除数据操作变得更为容易，并能令所耗时间达到最短。数据库管理系统必须在存储数据时保障其数据完整性；即保证数据是正确的，并且避免未经授权的用户访问它，也就是说数据是安全的。

关系型模型出现在 20 世纪 70 年代，对于终端用户而言，它能让数据操作变得更容易也更快速；而对于管理员来说，它能让数据的维护工作变得更加容易。该模型的核心概念是关系（relation），它可以表现为一个数据表，表中保存了所有的数据。数据可以表达为不同的类型，譬如字符串、数字、日期等等。每一个数据库表都由记录（record）或域（field）组成，其横向结构称之为行（row），纵向则称为列（column），类似于一个二维数组。数据库内的数据表是相互关联的；譬如，假定有一个名叫“school”的数据库，它可能由数据表“student”、“teacher”、“course”等构成。一个学生能选择一个老师开设的课程，而一个老师则可以开设一门或多门课程。用户可对 student、teacher 或 course 数据表单独地检索和操纵数据，也可基于某些公共关键域（common key field）把它们 join 起来。结构化查询语言（Structured Query Language，即 SQL）用于和关系型数据库“通话”，从而方便对数据库中各个表的数据进行检索、插入、更新和删除操作。

关系型数据库又可称为关系型数据库管理系统（DBMS），是当前最流行的数据库系统。目前已经出现了多种关系型数据库，其中包括 Oracle、Sybase、PostgreSQL Informix、SQL Server 以及 MySQL。

17.2.1 客户端 / 服务器型数据库

关系型数据库用到了所谓客户端 / 服务器（Client/Server）模型。目前，MySQL 是开源社区中最为流行的客户端 / 服务器型数据系统之一。

图 17-1 展示了一种客户端 / 服务器架构模型。用户可以进入命令行界面，然后启动一个 MySQL 客户端，并发布 MySQL 命令。客户端会生成一个请求给服务器，并由后者向数据库发送查询请求。数据库会把查询结果发回给服务器，这些结果最终将显示在客户端窗口中。在图中第二个场景中，向数据库发起连接的并不是命令行客户端，而是一段 Perl 脚本。它通过一种数据库接口发起连接，其效果类似于一种解释器。如果 Perl 脚本中含有能够连接某种数据库（这里就是 MySQL）的指令，那么在发起连接并选好某个数据库之后，Perl 程序便能通过 MySQL 服务器访问该数据库了。MySQL 服务器会从 Perl 程序接受请求（又称查询），并返回从数据库中收集到的信息。在图中第三个示例中，用户从浏览器（客户端）发起一个页面请求；浏览器向 Web 服务器（Apache、ISS）发起一个 HTTP 连接，并由后者负责接受和处理这些请求。如果需要启动一个 Perl 程序，Web 服务器便会使用公共网关接口（CGI）去启动 Perl 解释器，然后由 Perl 负责处理来自 HTTP 服务器的信息。Perl 可以将它们格式化后直接发送回 Web 服务器，或者也可在对数据库服务器发起请求时实现各项功能步骤，包括连接数据库、查询数据库、从数据库中获取结果。图 17-1 显示了 MySQL 客户端和 MySQL 服务器之间的客户端 / 服务器型关系，还有 Web 浏览器、Web 服务器上的 Perl 程序和 MySQL 数据库服务器之间的客户端 / 服务器型关系。学完本章内容后，读者将能够把信息从 Web 浏览器（客户端）发送到 Web 服务器，再从 Web 服务器转发给一段 Perl CGI 程序，并由后者连接到一个数据库服务器上，从而实现在 MySQL 数据库中检索和存储信息。

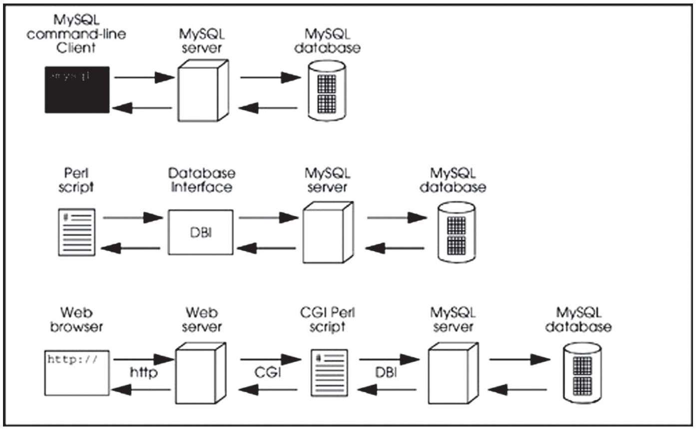


图 17-1 客户端 / 服务器模型

17.2.2 关系型数据库的组成部分

数据库是由什么组成的？组成一个关系型数据库管理系统的主要部分包括：

- 数据库服务器（Database server）
- 数据库（Database）
- 表（Table）
- 域（Field）
- 记录（Records）
- 主键（Primary Key）
- 样式表（Schema）

本章后面部分将详细讨论这些概念。图 17-2 说明了这些概念的相互关系。

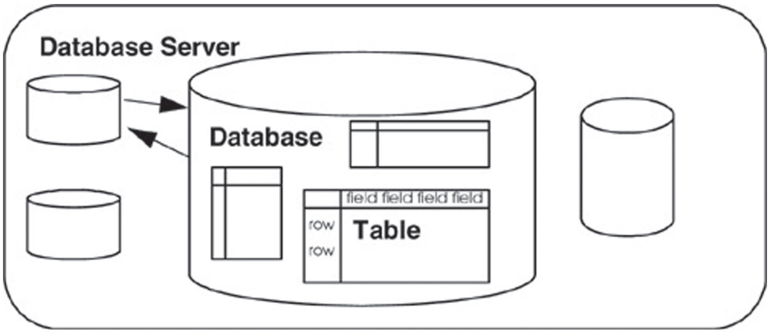


图 17-2 数据库服务器、数据库和表

数据库服务器。数据库服务器是实际负责运行数据库的服务器进程。它负责管理数据的存储、为用户提供访问途径、更新和删除记录，还能与其他服务器进行通信。数据库服务器一般都位于一台专用主机上，负责管理网络中的多台客户端；不过它也可部署为本地主机上只带有一个客户端的独立服务器；即，该客户端直接使用位于本地计算机（往往又称作“localhost”）上的 MySQL 服务器，而不必经过任何网络连接，这或许是学习如何使用 MySQL 的最佳方式。

如果读者正在使用的是 MySQL，其在 Windows 上的服务器进程是一个 mysql 服务，而在 Linux/UNIX 操作系统上则是 mysqld 进程。数据库服务器一般都遵循客户端 / 服务器模型。其前端就是客户端，用户可以坐在工作站前，发起数据库请求，并等待返回结果；其后端则是数据库服务器，它负责为用户提供访问权限、存储和处理数据、执行数据备份、乃至与其他服务器进行通信。发送到数据库服务器的请求还可以来自于一段程序，该程序将代表用户从 Web 页面或程序中发起请求。在后面的章节里，读者将首先学习如何从 MySQL 命令行客户端中产生请求；然后学习如何从一段 Perl 程序中使用 Perl 内建的函数连接数据库服务器，这些内建函数能向 MySQL 数据库服务器生成请求；最后还将学习如何从一个 Web 表单中生成请求，并将该请求发送到 Perl 程序内，进而转发给 MySQL。

数据库。 数据库是一组相关数据元素的集合，一般会与某种特定的应用相关联。一个公司可以为它的 HR 部门建立一个数据库，也可为销售部门建立另一个数据库，并为其电子商务应用程序建立第三个数据库，等等。图 17-3 列出了某一版 MySQL 中已安装的数据库。其中列出的数据库包括："mysql"、"northwind"、"phpmyadmin" 和 "test"。

表。 每个数据库都是由多个二维表组成的，这些表各自拥有惟一标识它的名字。事实上，关系型数据库会将它所有的数据存储到表中，一点也不能少。所有操作都是在表上执行的，这些操作也可用于生成其他的表，等等。

在设计一个数据库时，读者首先要做的决定之一就是它应当含有哪些表。常见的为某个组织提供的数据库可能会由顾客表、订单表和产品表构成。所有这些表格之间都通过某种方式相互关联。例如，顾客下了订单，而订单又分很多项目。尽管每个数据库都是独立存在的，但总体而言，这些表一起构成了一个数据库。图 17-4 列出了数据库"northwind"⊖中所有的表，该数据库是 Microsoft 公司提供的一个虚构的数据库，它作为一个模型帮助用户学习如何操纵数据库。（读者可以在华章网站（www.hzbook.com）中找到该数据库。）

```
+------------+
| Database   |
+------------+
| mysql      |
| northwind  |
| phpmyadmin |
| test       |
+------------+
```

图 17-3 MySQL 数据库

```
+----------------------+
| Tables_in_northwind  |
+----------------------+
| categories           |
| customercustomerdemo |
| customerdemographics |
| customers            |
| employees            |
| employeeterritories  |
| order_details        |
| orders               |
| products             |
| region               |
| shippers             |
| suppliers            |
| territories          |
| usstates             |
+----------------------+
```

图 17-4 northwind 数据库中的表

记录和字段。 表拥有自己的名字，它由一组行和列构成。它类似于一张电子表格，其每一行

⊖ Northwind Trader 这个示例数据库本来是随 Microsoft Access 提供的一个免费示例，不过它一样适用于 MySQL，可参考 http://www.flash-remoting.com/examples/。

又称作一个记录（record），由多个垂直的列组成，这些列亦可称为字段（field）。来自同一个表的所有行都拥有同样的一组列。“northwind”数据库的“shippers”表含有三个列与三个行，如图 17-5 所示。

```
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         1 | Speedy Express   | (503) 555-9831 |
|         2 | United Package   | (503) 555-3199 |
|         3 | Federal Shipping | (503) 555-9931 |
+-----------+------------------+----------------+
```

图 17-5 “northwind”数据库中“shippers”表的行（记录）与列（字段）

读者可以对关系型数据库执行两种基本操作。一种是检索其所有列的子集，另一种则是检索其所有行的子集。图 17-6 和 17-7 展示了这两种操作的示例。

```
mysql> select companyname from shippers;
+------------------+
| companyname      |
+------------------+
| Speedy Express   |
| United Package   |
| Federal Shipping |
+------------------+
```

图 17-6 检索列的子集

```
mysql> select * from shippers where companyname="Federal Shipping";
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         3 | Federal Shipping | (503) 555-9931 |
+-----------+------------------+----------------+
```

图 17-7 检索行的子集

记住：关系型数据库只能操纵表，其任何操作的执行结果也都是表，这些表又称作结果集合（result set）。表也是一种集合，它们自己就是行和列的集合。而数据库本身就是一组表的集合。

用户还可在两个表之间执行其他多种操作，并将它们当成集合来处理：譬如从两个表连接（join）信息，求多个表的卡笛尔乘积，求两个表的交集，将一个表加到另一个表上，等等。本章后面将介绍如何使用 SQL 语言在表上执行操作。

列 / 字段

列是任何一个数据库表都会提供的部分。列又称作字段，或者属性（attribute）。字段负责描述数据。每一个字段都拥有自己的名字。例如，“shippers”表拥有几个名叫“ShipperID”、“CompanyName”和“Phone”的字段。字段还可以描述其所含数据的类型。一个数据的类型可以是数字、字符、日期、时间戳（time stamp）等等。在图 17-8 中，ShipperID 是字段的名字，其数据类型是一个整数，并且该托运人 ID 的长度不会超过 11 个数字。数据类型有很多种，但有的时候某些数据类型是指定数据库系统专用的；譬如，MySQL 支持的数据类型就有可能与 Oracle 不同。读者将在后面学到更多有关 MySQL 数据类型的知识。

```
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
```

图 17-8 每个字段都拥有一个名字及其所保存数据的描述信息

行 / 记录

记录是表中的一个行。它在产品表中代表一个产品，在雇员表中代表一个雇员记录，依此类推。每个表都含有零或多个记录。图 17-9 表明在“shippers”表中有三个记录。

```
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         1 | Speedy Express   | (503) 555-9831 |
|         2 | United Package   | (503) 555-3199 |
|         3 | Federal Shipping | (503) 555-9931 |
+-----------+------------------+----------------+
3 rows in set (0.00 sec)
```

图 17-9　在“shippers”表中有三个记录

主键和索引

主键（primary key）是每个记录的一个惟一标识。例如，美国境内的每一位雇员都拥有一个社会安全号（Social Security Number），每个驾驶员都有自己的驾驶执照，而每辆汽车都有一个车牌号。这些标识符都是惟一的。在数据库的世界中，这些标识符又称作主键。虽然最好应该为每个表都提供主键，但并非所有的表都含有主键。在创建一张表时，同时也就确定了其主键。本章将在后面讨论数据库设计时详细介绍它。在图 17-10 中，“ShipperID”是“northwind”数据库中“shippers”表的主键。它是由一个数字组成的惟一 ID，每当向托运人列表中添加一个新的公司（记录）时，便

```
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
```

图 17-10　“ShipperID”是“shippers”表的主键

会自动递增这个数字。

当在某张表中搜索某个指定记录时，MySQL 必须在执行查询语句前载入所有的记录。除了主键之外，常常还会用到一个或多个索引，以便在表中搜索需频繁访问的行时提升其性能。索引（index）类似于书籍后面提供的目录，后者能帮助读者更快地找到所需主题，而不必搜索整本书。与书籍的目录类似，索引是一种指向表中特定记录的引用。

数据库样式表（Schema）。设计一个非常小的数据库并非难事，但是为某个基于 Web 的大型应用程序设计数据库则令人望而生畏。数据库的设计既是一门艺术，也是一门科学；它要求用户理解关系型模型是如何实现的，而这个主题则超出了本书介绍的范围。当讨论某个数据库的设计时，读者将会碰到“数据库 schama”一词，它指的是数据库的结构。它负责描述一个数据库的设计，类似于模板（template）或者蓝图（blueprint）；它能描述所有的表，以及如何组织其中的数据，但不含实际数据。图 17-11 描述了“northwind”数据库中各个表的样式表。

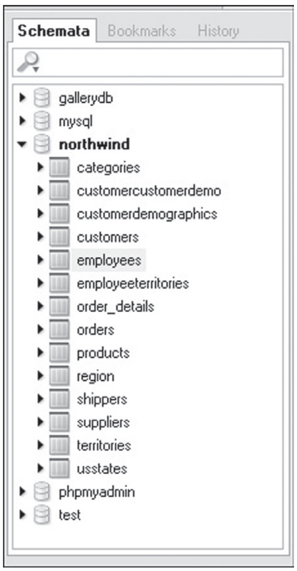


图 17-11　数据库样式表

17.2.3 通过 SQL（结构化查询语言）访问数据库

当把 Perl 的输出传送到浏览器时，浏览器只能理解像 HTML 和 XHTML 之类的标记语言（markup language），而这些语言标签会嵌入到 Perl 的 print 语句中，因此其输出内容才能显示为表单、图像、带格式的文本、颜色、表格等形式。同样，为了能与 MySQL 服务器进行通讯，读者的 Perl 脚本必须能说一种数据库可以理解的语言。该语言叫作 SQL。SQL 全称结构化查询语言（Structured Query Language），目前大多数支持多用户的关系型数据库都选用了该语言。它提供了一套语法和语言结构，能以标准化、跨平台 / 结构化的方式访问关系型数据库。正如英语拥有不同的方言（伦敦腔、美国英语、澳洲英语等等）一样，SQL 语言也有许多不同的版本。MySQL 使用的 SQL 版本遵循 ANSI（美国国家标准署）标准，也就是说它必须能支持该标准中定义的主要关键字（譬如 SELECT、UPDATE、DELETE、INSERT、WHERE 等等）。正如这些关键字的名称所示，SQL 是一种能够操纵数据库中的数据的语言。

如果读者尚不熟悉 SQL，请阅读附录 B，该附录提供了有关如何使用 SQL 语言的完整向导。此外在 Web 上也有许多写得很不错的入门指南，譬如 http://www.w3schools.com/sql/default.asp、http://sqlcourse.com/select.html 以及 http://www.1keydata.com/sql/sql.html。

准英语语法（English-like Grammer）。 当用户创建一条 SQL 语句时，它会向数据库产生一个请求或“查询”；该请求是一条语句，其结构类似于英语中的祈使句，譬如“Select your partner”、“Show your stuff”或“Describe that bully”。SQL 语句中的第一个单词是一个表示动作的英语动词，又称作命令，譬如 show、use、select、drop 等。命令后面跟有一组类似于名词的内容，譬如 show databses、use database 或 create database。语句中也可含有介词，譬如“in 或 from”；例如：show tables in database 或者 select phones from customer_table。SQL 语言还允许用户添加额外的条件子句，以便细化查询，譬如 select campanyname from suppliers where supplierid > 20。

当在一个查询中列出多项内容时，可以像英语一样用逗号隔开它们；例如在下面这个示例中，选中列表中的各个字段都是以逗号隔开的：select companyname, phone, address from suppliers ;。如果查询语句很长很棘手的话，读者可能希望把它们键入常用的编辑器中，因为一条查询会在执行完毕后自动消失。如果将查询保存到编辑器的话，读者就能将它剪切并粘贴回 MySQL 浏览器或命令行界面，而不必重新输入它。最重要的是，请读者确保查询的正确性，以免破坏某个重要的数据库。MySQL 提供了一个“test”数据库，专门用于这种验证工作。

以分号终止 SQL 语句。分号是结束每条查询语句的标准方法。某些数据库系统并不要求使用分号，但 MySQL 是要求的（USE 和 QUIT 命令除外）。如果用户忘记输入它，就会看到第二个提示符出现，同时会挂起执行，直到用户加上分号为止。

命名规范。使用良好的命名规范会让数据库和数据表显得更加易读。

例如，最好把表名设置为复数，而把字段 / 列名设置为单数形式。为什么呢？因为一个名叫“Shippers”的表一般持有多个 Shipper，而字段名则用于描述每一个 Shipper，因此是单数值，譬如“Company_Name”、“Phone”等等。表名或字段名的首字母一般都是大写的。

像“Company_Name”这样的组合式名称往往会通过下划线分隔开来，并且每个部分的首字母都是大写的。

在任何的数据库名中都不允许出现空格和斜杠。

保留字。所有语言都拥有一组保留字，它们对于该语言拥有特殊的含义。本章将用到大多数的

保留字。表 17-1 列出了 SQL 的保留字。（有关所有保留字的列表，请参阅 MySQL 文档。）

表 17-1　SQL 保留字

CREATE	ALTER
INSERT	SELECT
FROM	ON
ORDER BY	JOIN
CROSS JOIN	RIGHT JOIN
DROP	DELETE
UPDATE	SET
INTO	WHERE
GROUP BY	LEFT JOIN
FULL JOIN	AND
LIMIT	LIKE
OR	AS

大小写敏感。如果用户正在使用 UNIX，则数据库名和表名都是大小写敏感的；如果是 Windows 的话则是不敏感的。规范往往都是把数据库及其数据表都以小写方式命名的。

SQL 对大小写则并不敏感。例如，下列 SQL 命令是等效的：

```
show databases;
SHOW DATABASES;
```

尽管 SQL 命令并不区分大小写，但是按照规范的规定，SQL 关键字应当全部大写，而字段名、表名和数据库名只有首字母为大写，以便让命令更为清晰。

```
SELECT * FROM Persons WHERE FirstName='John'
```

如需使用 LIKE 和 NOT LIKE 命令执行模式匹配的话，则在 MySQL 中这些查询的模式将是大小写敏感的。

结果集。结果集（result set）只不过是另一个表，它含有某个 SQL 查询的执行结果。大多数数据库系统甚至允许用户使用函数在结果集上执行操作，譬如 Move-To-First-Record、Get-Record-Content、Move-To-Next-Record 等等。在图 17-12 中显示的结果集是由一条要求 mysql 显示“shippers”表中所有字段的查询语句创建的。

```
mysql> show fields in shippers;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

图 17-12　结果集只是由查询产生的一张表

17.3 MySQL 入门

17.3.1 为何选用 MySQL

MySQL 是一种开放源码的⊖、全功能的关系型数据库管理系统，已经移植到绝大多数平台上，包括 Linux、Windows、OS/X、HP-X、AIX 等。MySQL 是可移植（能运行在 Linux、Windows、OS/X、HP-X 等系统上）、快速、可靠、可扩展的，并且非常易于使用。

它号称已经安装在全世界范围内的数千万台计算机上，甚至还包括南极洲！它拥有两种版本：一种需要用户购买商业化许可证，另一种则是免费的。免费的意思是：用户可以在不拷贝、修改或传播 MySQL 软件的前提下把 MySQL 用在任意应用程序中。MySQL 支持多种 API（应用编程接口），包括 Perl、PHP、TCL、Python、C/C++、Java 等。

当使用 MySQL 时，读者会碰到一些类似于名字的术语。表 17-2 能帮助读者弄清楚这些词汇的用法。

表 17-2 MySQL 中的术语，等等

mySQL	充当数据库管理系统的实际软件
mysqld	MySQL 守护进程，或服务器进程
mysql monitor	发出 MySQL 命令时的监视器（命令行解释器）
mysql	MySQL 用于管理访问权限的数据库的名字
mysqladmin	一种 MySQL 实用程序，用于管理数据库

17.3.2 安装 MySQL

这里假定读者已经安装并启动了一个数据库服务器。下载和安装 MySQL 往往是一个非常直观的过程。读者可以从 mysql.com Web 站点获得 MySQL，也可使用像 XAMMP 或 WAMP 之类的集成应用程序获得它。

XAMPP（针对 Windows、Linux、Mac OS 和 Solaris）是一种免费的、易于安装的 Apache 发布包，其中含有 MySQL、PHP 与 Perl。读者只需下载它，然后解压并启动它便可。详见 http://www.apachefriends.org/en/xampp.html。

如需获得完整的安装帮助，请访问如图 17-13 所示的 Web 页面。

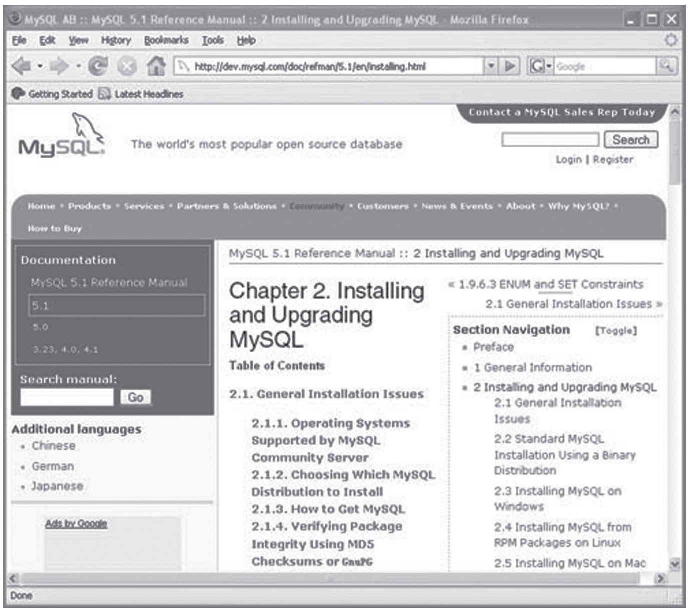


图 17-13 MySQL 安装文档

17.3.3 连接 MySQL

MySQL 数据库系统用到了第 17.2.1 节所描述的客户端 / 服务器模型。读者是一个连接到数据库上的客户端，它可以来自命令行、图形化界面，亦可来自某段程序。在学习如何从一段 Perl 程序中连接数据库之前，

⊖ MySQL 只能在 100％遵循 GPL 许可证的应用程序中使用。详见 http://www.mysql.com/company/legal/licensing/opensource-licence.html。

我们首先介绍如何使用 MySQL 的命令行客户端。

MySQL 命令行客户端由 MySQL 安装包提供，它适用于任何情况。它是一个 mysql.exe 程序，位于 MySQL 安装包的 bin 文件夹下。

为了运行这个命令行应用程序，读者首先应当启动命令行提示符。在 Windows 中，进入开始菜单，选择“运行 ..”选项，然后在“运行”窗口中键入 cmd。在 Mac OS X 上，请进入 Finder 中的 Applications 文件夹，然后转到 Utilities，在这里读者可以找到 Terminal 应用程序。读者应当转到 MySQL 安装位置，并找到其中的 bin 文件夹。在 UNIX 中，请在某个终端窗口中的 shell 提示符后面键入命令。

MySQL 客户端可执行文件一般都位于 bin 文件夹中。

为了使用上述客户端连接某个数据库，读者需要输入类似于下面这一行的内容（参见图 17-14）：

```
mysql --user=root -password=my_password --host=localhost
```



图 17-14　MySQL 客户端

不论选用哪种客户端，读者都需要指定用户名以及连接到的主机名。大多数配置都要求用户提供口令，不过如果读者只是自己用用，就无需提供口令了。读者还可以在选项中指定一个默认的数据库。

当成功连接到数据库之后，读者将看到一个 mysql> 提示符，而不是原来的标准 DOS/UNIX 提示符。这就意味着读者现在是向 MySQL 数据库服务器发送命令，而不是向本地的计算机操作系统。

在 MySQL 控制台中编辑键。 MySQL 支持对输入行的编辑。上箭头和下箭头键允许用户在前后输入行之间上下移动，而左箭头与右箭头则负责在一行内左右移动。退格（Backspace）键和删除（Delete）键用于删除一行内的字符，以便在光标位置键入新的字符。如需提交已经编辑好的行，请按下回车（Enter）键。针对 UNIX 用户，MySQL 还支持制表符补全（tab completion）功能，允许用户只输入某个关键字或标识符的部分内容，然后通过制表符（Tab）键补全它。

设置口令。 当下载 MySQL 或进行安装时，读者可能需要为用户“root”输入一个口令。尽管并不强制要求用户创建口令，但为了自己数据库的安全，还是设置一个口令为好。如需设置口令，请进入 MySQL 控制台并键入下列 MySQL 命令。请把其中的“MyNewPassword”替换为自己的口令。

```
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('MyNewPassword ') ;
```

设置完口令之后，用户应停止 MySQL 服务器并以普通模式重新启动它。如果该服务器是以服务形式运行的话，请从 Windows Services 窗口启动它。如果是手动启动服务器，用户则可使用自己

常用的任何相关命令。然后便可使用新口令连接了。

在 MySQL 提示符设置口令

示例 17.1

```
1 $ mysql -u root
ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using
password: NO)
$
$ mysql -u root -p
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 91 to server version: 5.0.21-community-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

17.3.4 图形化用户界面

MySQL 查询浏览器。 MySQL 查询浏览器（MySQL Query Browser）来自 mysql.com 的一种图形化用户界面（GUI）客户端，它用于连接 MySQL 数据库服务器。只要下载它并对照简单的安装向导按部就班安装，用户便可在 Windows 的开始菜单中启动该应用程序。

然后，MySQL 查询浏览器会显示一个连接对话框（参见图 17-15）。用户必须指定所要连接的 MySQL 服务器、该服务器授权所需的用户身份、该服务器所在计算机（以及监听端口）、以及将要使用的默认数据库（又叫“Schema”）。用户还可在需要时设置其他一系列附加选项。

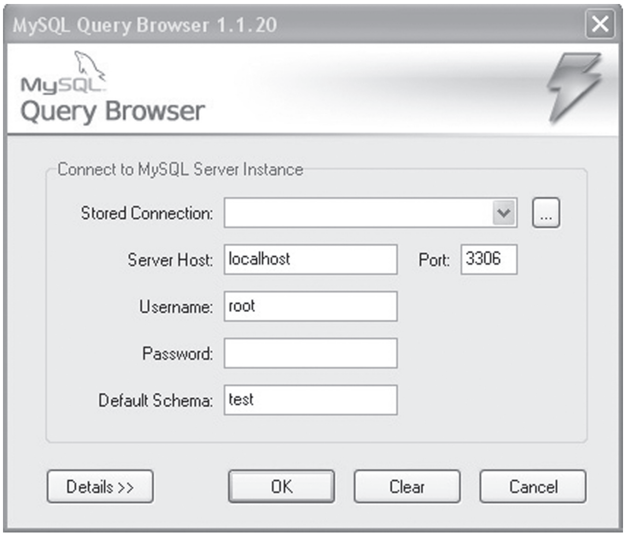


图 17-15 MySQL 查询浏览器

为了发起查询请求，用户必须选定一个默认的数据库。虽然在连接到服务器后也可以设置默认服务器，不过最好是从连接对话框中设置它，以便为后续连接节省时间。

这里填入的信息非常类似于命令行客户端：用户名、口令、以及运行数据库服务器的服务器计

算机。读者可以选择性地输入数据库名和端口号（MySQL 默认为 3306），还能在 Stored Connection 部分中将连接信息保存为书签。

通过使用位于应用窗口右侧的树型导航结构，用户还可在 MySQL 查询浏览器中转向不同的数据库，参见图 17-16。

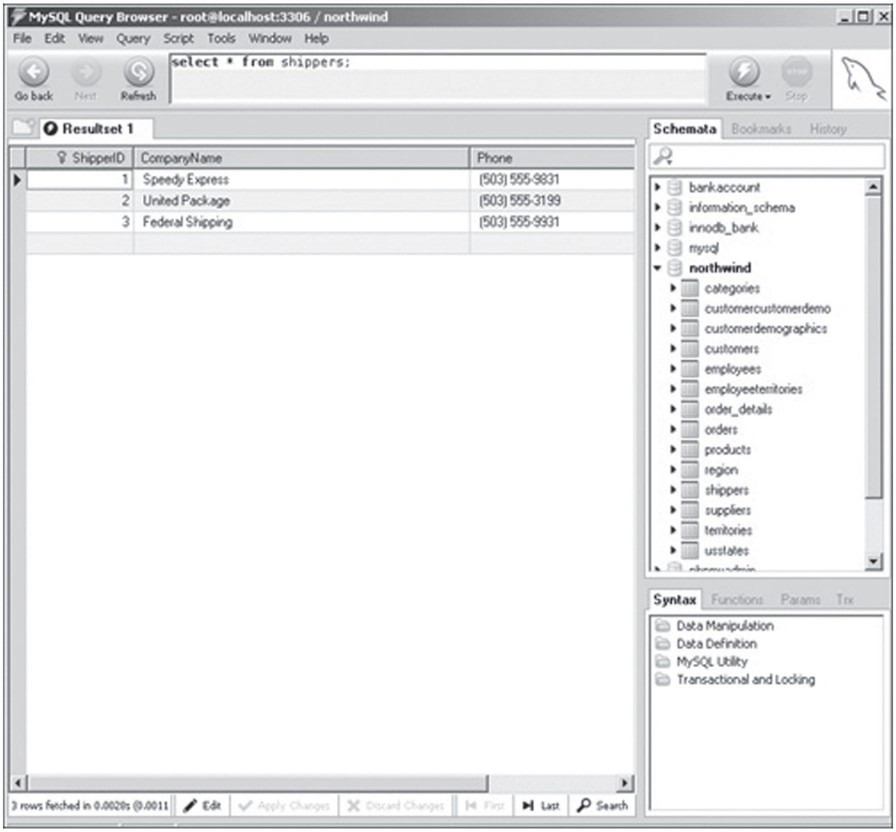


图 17-16 MySQL 查询浏览器

phpMyAdmin 工具

phpMyAdmin 工具以 PHP 语言编写，能在 Web 上处理对 MySQL 的管理操作（参见图 17-17）。它可以用于创建或删除数据库、操纵表和字段、执行 SQL 语句、管理字段上的键、管理权限，并能把数据导出为不同的格式。详见 http://www.phpmyadmin.net/home_page/index.php。

MySQL 权限系统。以驾驶执照为例，“认证（authentication）”负责通过检查照片和有效期来判断你是否真的拥有这本驾驶执照，而“授权（authorization）”则负责检查你有权驾驶哪些类型的汽车，譬如轿车、大卡车或校车。

与之类似，MySQL 权限系统的首要功能是在前面所述的命令行或图形化客户端中对用户与口令进行认证，以保证其有权连接到给定的主机上。MySQL 权限系统的第二个目的则是确定某个已经连接到数据库上的用户有权做哪些事情。例如，某些用户可能只有权选取或查看某个指定表中的数据。当安装完 MySQL 后，创建的 MySQL 数据库中便带有一些称作授权表（grant table）的表，

它们定义了最初的用户账户及其权限。其中第一个账户是用户“root”，又称作超级用户。超级用户能够做任何事情，也就是说，一旦某个用户以 root 身份登录数据库，就会拥有所有的权限。root 账户一开始不带任何口令，因此任何人都能以超级用户身份登录。其他的账户类型则是匿名用户（anonymous-user）账户，它们也不带有口令。对于 root 和 anonymous 账户，Windows 会合用一个口令，而 UNIX 则提供了两个不同的口令。这两种方式都能避免出现安全问题。在启动 MySQL 后，用户首先应当做的就是为 root 账户和 anonymous 账户设定口令。

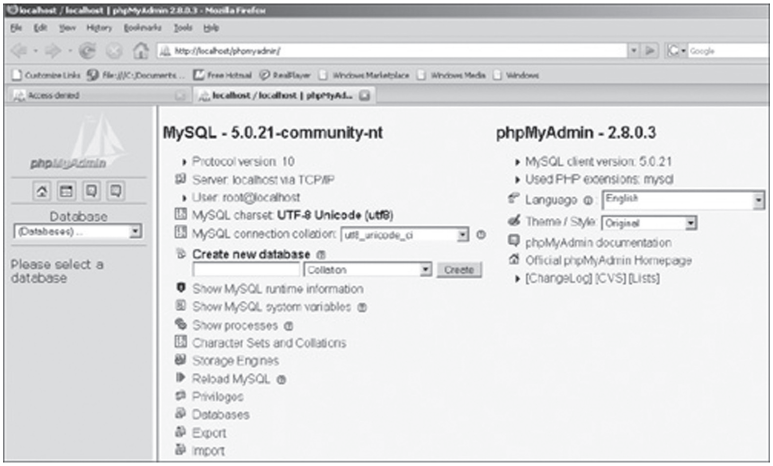


图 17-17　phpMyAdmin 工具

如需执行管理操作，用户必须以 root 身份访问服务器。mysqladmin 实用程序能够用于创建密码以及执行其他 MySQL 管理任务。在下面这个示例中，它就用于为用户“root”创建密码。

示例 17.2

```
1  $ mysqladmin -u root -h localhost password quigley1

2  $ mysql -uroot -hlocalhost -pquigley1
   Welcome to the MySQL monitor.  Commands end with ; or \g.
   Your MySQL connection id is 29 to server version: 5.0.21
community-nt

   Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

解释

1. mysqladmin 程序用于在本地主机上为用户 root 创建密码。其密码是“quigley1”。
2. 用户 root 登录数据库服务器。-u 开关后面跟有用户名或登录名(在 -u 和用户名之间没有空格)。这里用户是以“root”身份登录的。与之类似，-p 开关后面跟有实际的密码，在这里就是“quigley1”。如果没有提供密码的话，系统将提示用户输入密码。

17.3.5　寻找数据库

数据库服务器维护了一个含有可用数据库的列表。通过在 mysql 提示符下执行 show 命令，便可将该列表以一张表的形式显示出来，如下面的示例所示。一般而言，在安装完 MySQL 后，它会

带有两个数据库：test 和 mysql。test 是一个空数据库，用于练习和测试各种特性。用户一般不必拥有什么特殊权限就能使用 test 数据库。mysql 数据库是一个特殊数据库，MySQL 服务器用它保存各种访问权限。除非需要管理权限，否则就无需关心这个数据库。请参阅 MySQL 使用手册中的“GRANT”命令。

示例 17.3

```
1 mysql -uroot -pquigley1
  Welcome to the MySQL monitor.  Commands end with ; or \g.
  Your MySQL connection id is 5 to server version: 4.1.11-nt

  Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
2  mysql> show databases;
  mysql> show databases;
  +--------------------+
  | Database           |
  +--------------------+
  | information_schema |
  | mysql              |
  | phpmyadmin         |
  | test               |
  +--------------------+
  rows in set (0.00 sec)
 mysql>
```

解释

show databases 命令列出了该服务器上的所有数据库。通常情况下，在安装好 MySQL 后，用户就能得到 mysql 数据库和 test 数据库。test 数据库只用于测试目的，并且是空的。mysql 数据库则含有所有 MySQL 服务器权限信息。

创建和删除数据库。创建数据库非常简单。设计数据库是另一项内容，它取决于用户需求，以及用户将用哪种模型组织数据。在最小的数据库中，读者也必须创建表。下面一节将讨论如何创建和删除数据库与表。假定用户已授予了创建数据库的权限，便可在 mysql 命令行或使用 mysqladmin 工具来创建之，如下所示：

示例 17.4

```
1     mysql> create database my_sample_db;
      Query OK, 1 row affected (0.00 sec)

2     mysql> use my_sample_db;
      Database changed

3     mysql> show tables;
      Empty set (0.00 sec)

4     mysql> create table test(
      -> field1 INTEGER,
      -> field2 VARCHAR(50)
      -> );
      Query OK, 0 rows affected (0.36 sec)
5     mysql> show tables;
      +----------------------+
      | Tables_in_my_sample_db |
```

```
        +------------------------+
        | test                   |
        +------------------------+
        1 row in set (0.00 sec)

6       mysql> drop table test;
        Query OK, 0 rows affected (0.11 sec)

7       mysql> drop database my_sample_db;
        Query OK, 0 rows affected (0.01 sec)
```

解释

1. 该命令创建一个名叫 my_smaple_db 的数据库。
2. 创建好数据库并不代表读者已经身在其中。如需进入新的数据库，请执行 use 命令。
3. show 命令列举数据库中的所有表。该数据库是空的。
4. 为 my_smaple_db 数据库创建一个名为 test 的表。在创建该表时，定义两个列 field1 和 field2。为每个字段指定它所保存的数据类型；field1 将保存纯数字，而 field2 会保存最大长度为 50 个字符的一个字符串。
5. show 命令列出数据库中的所有表。
6. drop table 命令负责销毁一张表及其所有内容。
7. drop database 命令负责销毁一个数据库及其所有内容。

17.3.6 基本命令入门

示例 17.5

```
(At the MySQL Client Prompt)
1 mysql> select database();
+------------+
| database() |
+------------+
| northwind  |
+------------+
1 row in set (0.00 sec)
2 mysql> select now();
+---------------------+
| now()               |
+---------------------+
| 2007-05-12 16:09:57 |
+---------------------+
1 row in set (0.00 sec)
```

下面一节中的示例将展示如何从 MySQL 客户端中执行 SQL 命令。这些示例并不奢求涵盖 MySQL 支持的所有 SQL 语句，而是说明了一些基本的语法，以便创建与删除数据库和表，并说明如何在数据库表中插入、删除、编辑、更改和选取数据。如需了解完整的 MySQL 功能描述，请读者访问 http://dev.mysql.com/doc/（参见图 17-18）。

借助 MySQL 创建数据库。 现在读者可以创建一个数据库了。该数据库名叫 sample_db。CREATE DATABASE 命令负责创建数据库，而 SHOW DATABASES 语句则说明了用户现在正位于另一个数据库中。（读者还可使用 mysqladmin 命令来创建和删除数据库。）

龍睿 · LoRui

本 PDF 书签由 龙睿 • LoRui 制作。

www.LoRui.com

来看看这些东东……

- ✧ Yahoo!疯了！
- ✧ HP笔记本激活Windows 7

域名转让：

- ✓ DaNanPing.com（大南平）
- ✓ 15500.net
- ✓ Gn7c.com
- ✓ DaLongNan.com（大陇南、大龙南）
- ✓ 51perl.com（无忧 Perl）
- ✓ LoRui.com

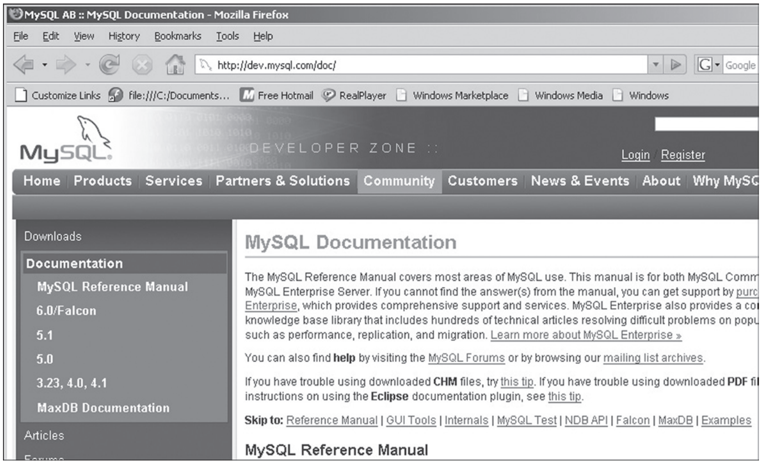


图 17-18 MySQL 文档页面

示例 17.6

```
1  mysql> CREATE DATABASE sample_db;
Query OK, 1 row affected (0.03 sec)

2 mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| northwind          |
| phpmyadmin         |
| sample_db          |
| test               |
+--------------------+
6 rows in set (0.00 sec)
```

解释

1. CREATE DATABASE 语句允许用户创建一个数据库。创建数据库操作并不能让用户进入该数据库。USE 语句将让用户在这个新数据库中工作，正如后面示例所示。
2. SHOW DATABASES 语句能列出当前可用的所有 MySQL 数据库。数据库 sample_db 刚刚创建。

借助 MySQL 选择数据库。 当创建好数据库之后，在使用它之前，必须首先打开该数据库。该项操作可通过 USE 语句来完成。现在，我们就有一个待处理的数据库了。

示例 17.7

```
mysql> USE sample_db;
Database changed
```

在数据库中创建表。 当创建好数据库之后，是时候创建一些表了。在实际情况下，关系型数据库的设计过程涉及一组为表结构添加逻辑限制的规则、一个叫作“规范化（normalization）”的处理过程、以及超出本书范围的其他一些方面。在示例数据库中，我们将创建一张表，并向它添加数

据，以便告诉读者应当怎么做。数据类型定义了表中每一个字段的结构。CREATE TABLE 语句能定义每一个字段、它的名字以及它的数据库类型。

数据类型。首选，我们必须决定哪些种类的数据将能保存到表中：文本、数字、日期、照片、金额，等等。同时还要决定这些用于保存数据的字段（列）的名称。MySQL 指定了一系列数据类型，能够描述保存在数据库中的所有类型的数据。大多数 MySQL 数据类型都在表 17-3 中予以列出。

表 17-3　SQL 数据类型

数据类型	描　述
数字型	
TINYINT	非常小的数字；适合表达年龄值。如果带 UNSIGNED 子句，它能保存 0 到 255 之间的值；否则其范围则是从 -128 到 127
SMALLINT	适合表示范围为 0 到 65535（UNSIGNED）或从 -32768 到 32767 之间的数字
MEDIUMINT	带 UNSIGNED 子句时，范围是从 0 到 16777215；不带时，范围是从 -8388608 到 8388607
INT	带 UNSIGNED 的整数范围是从 0 到 4294967295；不带时则是从 -2147483648 到 2147483647
BIGINT	大数字（从 -9223372036854775808 到 9223372036854775807）
FLOAT	浮点数（单精度）
DOUBLE	浮点数（双精度）
DECIMAL	以字符串形式表示的浮点数
文本型	
CHAR(x)	x 范围从 1 到 255。持有一个长度固定的字符串（可含有字母、数字与特殊字符）。其固定长度值位于括号中
VARCHAR(x)	x 范围从 1 到 255。持有一个长度可变的字符串（可含有字母、数字与特殊字符）。其最大长度值位于括号中
TINYTEXT	一小段文本，大小写敏感
TEXT	稍微长一些的文本，大小写敏感
MEDIUMTEXT	中等长度的文本，大小写敏感
LONGTEXT	较长的文本，大小写敏感
TINYBLOB	Blob 的意思是一段二进制大型对象（Binary Large Object）。在执行大小写敏感的搜索操作时，应当使用 blob 类型
二进制型	
BLOB	稍微大一些的 blob，大小写敏感
MEDIUMBLOB	中等大小的 blob，大小写敏感
LONGBLOB	较为大型的 blob，大小写敏感
ENUM	枚举数据类型具有固定的长度，每一列只能含有给定集合中的单个值。这些值都放在 ENUM 声明后面的括号中。例如在一个战争状态（m_status）列中，可有 ENUM("Y" , "N")
日期型	
DATE:YYYY-MM-DD	四位年份后面跟有两位的月份和日期；例如，20071030
TIME: hh:mm:ss	小时 : 分钟 : 秒钟
DATETIME	YYYY-MM-DD hh:mm:ss（日期和时间之间通过空格隔开）
TIMESTAMP	YYYYMMDDhhmmss
YEAR	YYYY（四位年份值）；例如 2007

示例 17.8

```
1  mysql> CREATE TABLE teams(
    -> name varchar(100) not null,
    -> wins int unsigned,
    -> losses int unsigned);
Query OK, 0 rows affected (0.09 sec)

2 mysql> SHOW TABLES;
+---------------------+
| Tables_in_sample_db |
+---------------------+
| teams               |
+---------------------+
1 row in set (0.00 sec);

3 mysql> DESCRIBE teams;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| name   | varchar(100)     | NO   |     |         |       |
| wins   | int(10) unsigned | YES  |     | NULL    |       |
| losses | int(10) unsigned | YES  |     | NULL    |       |
+--------+------------------+------+-----+---------+-------+
| teams               |
+---------------------+
1 row in set (0.00 sec)
```

解释

1. CREATE TABLE 语句在数据库中创建一张表。该表名为 teams。它含有三个字段：name、wins、losses。字段 name 含有最大长度为 100 个字符的一个可变长字符串；wins 和 losses 字段则分别含有无符号的整数。
2. SHOW 命令列出数据库中所有的表。
3. DESCRIBE 命令能够描述表的结构；即，各个字段的名称，以及每个字段保存的数据类型。请注意，这里的 name 字段不能为“NULL”；如果没有向 wins 与 losses 字段赋值的话，这两个字段的默认值将是 NULL；例如，如果某支球队尚未参加任何比赛，它的 wins 和 losses 字段就会赋值为 NULL。

借助主键添加另一张表。 在下面这个示例中，我们将创建另一张表，并添加一个主键。主键用于惟一标识数据库中的记录。用户的登录名、UID、账户编号或者许可证都是惟一 ID 的示例。主键是一种惟一的索引，其所有关键列（key column）都必须定义为 NOT NULL．如果没有显式地把它们定义为 NOT NULL 的话，MySQL 将会隐式地（悄悄地）声明它们。一张表只能含有一个主键。

示例 17.9

```
1  mysql> CREATE TABLE coaches(
    -> id INT NOT NULL AUTO_INCREMENT,
    -> name VARCHAR(75),
    -> team VARCHAR(100),
    -> title VARCHAR(50),
    -> start_date date,
    -> PRIMARY KEY(id));
```

```
   Query OK, 0 rows affected (0.38 sec)

2  mysql> DESCRIBE coaches;
   +------------+--------------+------+-----+---------+---------------
   | Field      | Type         | Null | Key | Default | Extra
   +------------+--------------+------+-----+---------+---------------
   | id         | int(11)      | NO   | PRI | NULL    |auto_increment
   | name       | varchar(75)  | YES  |     | NULL    |
   | team       | varchar(100) | YES  |     | NULL    |
   | title      | varchar(50)  | YES  |     | NULL    |
   | start_date | date         | YES  |     | NULL    |
   +------------+--------------+------+-----+---------+---------------
   5 rows in set (0.00 sec)
3  mysql> SHOW TABLES;
  +---------------------+
  | Tables_in_sample_db |
  +---------------------+
  | coaches             |
  | teams               |
  +---------------------+
  2 rows in set (0.00 sec)
```

解释

1. 在创建 coach 表时，将 id 字段设定为它的主键。该字段将用于惟一标识某个特定的教练员。
2. 新表的结构显示其 id 字段是一个主键，每当添加新的教练时，MySQL 会自动把该字段值递增 1。
3. 现在该数据库含有两张表，一张叫作 teams，另一张名叫 coaches。

向表中插入数据。 SQL INSERT 语句能把新记录添加到表中。在插入数据时，请务必保证按照数据存储的实际顺序来为各个字段提供对应值；否则 SQL 将会发送一条错误消息。在下面的示例中，为了插入数据，用户既可通过 SET 子句为特定字段赋值，也可用 VALUES 列表指定需要的值，还可简单地按顺序依次列出每一个字段的值。

示例 17.10

```
1  mysql> INSERT INTO teams
    -> set name='Fremont Tigers',
    -> wins=24,
    -> losses=26;
   Query OK, 1 row affected (0.00 sec)

2  mysql> INSERT INTO teams
    -> set name='Chico Hardhats',
    -> wins=19,
    -> losses=25;
   Query OK, 1 row affected (0.00 sec)

3  mysql> INSERT INTO teams values
    -> ('Bath Warships',32,3);
   Query OK, 1 row affected (0.00 sec)
4  mysql> INSERT INTO teams values
    -> ('Bangor Rams', 22, 24);
   Query OK, 1 row affected (0.00 sec)

5  mysql> SELECT name FROM teams;
```

```
    +-----------------+
    | name            |
    +-----------------+
    | Fremont Tigers  |
    | Chico Hardhats  |
    | Bath Warships   |
    | Bangor Rams     |
    +-----------------+
    4 rows in set (0.00 sec)

6  mysql> INSERT INTO coaches values
    -> (" ",'John Doe','Chico Hardhats','Head Coach', 20021210);
    Query OK, 1 row affected, 1 warning (0.05 sec)

7  mysql>  INSERT INTO coaches values
       -> (" ", 'Jack Mattsone','Chico Hardhats','Offensive Coach' ,
'20041005');
     Query OK, 1 row affected, 1 warning (0.00 sec)

8  mysql> INSERT INTO coaches(name,team, title,start_date)
        -> values( 'Bud Wilkins', 'Fremont Tigers', 'Head Coach',
'19990906');
     Query OK, 1 row affected (0.03 sec)

9  mysql>  INSERT INTO coaches(name, team, title,start_date)
        ->values( 'Joe Hayes', 'Fremont Tigers', 'Defensive Coach',
     '19980616');
     Query OK, 1 row affected (0.02 sec
```

解释

1. 在 INSERT 语句中，使用 SET 子句指派字段和值。对于那些没有出现在 SET 子句中的字段，MySQL 会把它们赋值为默认值。
2. 再次使用 SET 子句指派字段和值。
3. 在本例中，INSERT 语句含有一个 VALUES 列表，它能以创建表时指定的顺序依次为每一个字段赋值。如果对顺序情况不大确定，可使用前面示例所述的 DESCRIBE 语句来查看。
4. 在这个示例中，为一条新记录再次重复 VALUES 列表。请注意，此处日期是以 yyyy-mm-dd 的形式而插入的。
5. SELECT 语句能够显示插入到表中的所有球队的名字。

6.7.8.9. 借助 VALUES 列表插入更多的记录。

从表中选择数据——SELECT 命令。 最常用的一个 SQL 命令是 SELECT，负责执行一条查询。SELECT 命令用于根据某些标准从一张表中检索数据。它指定了一组以逗号隔开的待检索字段，以及一个指定待访问数据表的 FROM 子句。其执行结果将保存在一个结果表中，又称作结果集。符号 * 用于表示所有的字段。

按列选择。 下面这个示例将会检索指定列的数据，其中每个列（字段）都以逗号隔开。

示例 17.11

```
1  mysql> SELECT name FROM teams;
+-----------------+
| name            |
```

```
+-----------------+
| Fremont Tigers |
| Chico Hardhats |
| Bath Warships  |
| Bangor Rams    |
+-----------------+
4 rows in set (0.00 sec)

2  mysql> SELECT name, wins FROM teams;
  +----------------+------+
  | name           | wins |
  +----------------+------+
  | Bangor Rams    |   22 |
  | Bath Warships  |   32 |
  | Fremont Tigers |   24 |
  | Chico Hardhats |   19 |
  +----------------+------+
  4 rows in set (0.00 sec)

3  mysql> SELECT id, name, title FROM coaches;
  +----+---------------+-----------------+
  | id | name          | title           |
  +----+---------------+-----------------+
  | 1  | John Doe      | Head Coach      |
  | 2  | Jack Mattsone | Offensive Coach |
  | 3  | Bud Wilkins   | Head Coach      |
  | 4  | Joe Hayes     | Defensive Coach |
  +----+---------------+-----------------+
  4 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出 name 字段的所有值。
2. SELECT 语句从名叫 teams 的表中检索出 name 字段与 wins 字段的所有值。字段名列表之间通过逗号予以隔开。
3. SELECT 语句从 coaches 表中检索出 id 字段、name 字段和 title 字段的所有值。

选择所有的列。 * 是一个通配符，用于表示一张表内所有的列。

示例 17.12

```
1  mysql> SELECT * FROM teams;
+----------------+------+--------+
| name           | wins | losses |
+----------------+------+--------+
| Fremont Tigers |   24 |     26 |
| Chico Hardhats |   19 |     25 |
| Bath Warships  |   32 |      3 |
| Bangor Rams    |   22 |     24 |
+----------------+------+--------+

2  mysql> SELECT * FROM coaches;
+----+---------------+----------------+-----------------+-----------+
| id | name          | team           | title           | start_date|
+----+---------------+----------------+-----------------+-----------+
|  1 | John Doe      | Chico Hardhats | Head Coach      | 2002-12-10|
```

```
|  2 | Jack Mattsone | Chico Hardhats | Offensive Coach | 2004-10-05|
|  3 | Bud Wilkins   | Fremont Tigers | Head Coach      | 1999-09-06|
|  4 | Joe Hayes     | Fremont Tigers | Defensive Coach | 1998-06-16|
+----+---------------+----------------+-----------------+-----------+
4 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出所有的字段及其值。
2. SELECT 语句从名叫 coaches 的表中检索出所有的字段及其值。

WHERE 子句。 WHERE 子句是可选的，它能根据某些条件（称作选择场景）来指定选择那些数据值或行。SQL 提供了一组运算符，以便指明待设定的条件。参见表 17.4。

表 17-4　SQL 运算符

运算符	描述	示例
=	等于	where country = 'Sweden'
<> , !=	不等于[a]	where country <> 'Sweden'
>	大于	where salary > 50000
<	小于	where salary < 50000
>=	大于或等于	
<=	小于或等于	
IS [NOT] NULL	是 NULL（非值）或不是 NULL	where birth = NULL
BETWEEN	位于某个范围之间	where last_name BETWEEN 'Doe' AND 'Hayes'
LIKE	搜索类似于模式的值	where last_name LIKE 'D% '
NOT LIKE	搜索不类似于模式值	where country NOT LIKE 'Sw%'
! , NOT	逻辑非，否定	where age ! 10;
|| , OR	逻辑或	where order_number>10 || part_number= 80
&& , AND	逻辑与	where age > 12 && age <21
XOR	异或	where status XOR

a. 在某些版本的 SQL 中，运算符 <> 亦可写成 !=。

示例 17.13

```
1  mysql> SELECT name, wins FROM teams WHERE wins > 25;
  +-------------------+------+
  | name              | wins |
  +-------------------+------+
  | Bath Destroyers   |   34 |
  | Portland Penguins |   28 |
  +-------------------+------+

2  mysql> SELECT name FROM teams WHERE losses < wins;
  +---------------+
  | name          |
  +---------------+
  | Bath Warships |
```

```
  +----------------+
  1 row in set (0.03 sec)
3  mysql> SELECT name, title FROM coaches WHERE team = 'Chico Hardhats';
  +----------------+-----------------+
  | name           | title           |
  +----------------+-----------------+
  | John Doe       | Head Coach      |
  | Jack Mattsone  | Offensive Coach |
  +----------------+-----------------+
  2 rows in set (0.00 sec)

4 mysql> SELECT name FROM coaches WHERE name LIKE 'J%';
  +----------------+
  | name           |
  +----------------+
  | John Doe       |
  | Jack Mattsone  |
  | Joe Hayes      |
  +----------------+
  3 rows in set (0.00 sec)

5  mysql> SELECT name FROM teams WHERE wins > 10 && losses < 10;
  +----------------+
  | name           |
  +----------------+
  | Bath Warships  |
  +----------------+
  1 row in set (0.00 sec)

6  mysql> SELECT name FROM coaches WHERE id BETWEEN 1 AND 3;
  +----------------+
  | name           |
  +----------------+
  | John Doe       |
  | Jack Mattsone  |
  | Bud Wilkins    |
  +----------------+
  3 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出胜利次数大于 25 的球队名称。
2. SELECT 语句从名叫 teams 的表中检索出败北次数小于胜利次数的球队名称。
3. 当教练所在球队等于字符串‘Chico Hardhats’时，SELECT 语句就从名叫 coaches 的表中检索出教练姓名和头衔。此处的字符串必须位于引号内，可以是单引号或者双引号，其匹配过程必须是精确的。
4. 如果教练的名字是以 'J' 开头的话，SELECT 语句就从名叫 coaches 的表中检索出这些教练的名字。% 符号是一个通配符，表示跟在 'J' 后面的任意字符。
5. 当胜利次数大于 10 且败北次数小于 10 时，SELECT 语句就会检索出这些球队的名称。&& 又称作逻辑与（logical AND）。其两侧语句都必须为真，否则就不会选到任何内容。
6. SELECT 语句从名叫 coaches 的表中检索出其教练 ID 介于 1 和 3 之间的教练的姓名。BETWEEN 子句能创建一个用于选择的范围。

对表进行排序。通过使用 ORDER BY 子句，用户便能按照指定的次序显示查询的结果。各行可以按升序（ASC，默认情况）或降序（DESC）进行排列，排序的值既可以是字符串，也可以是数字。用户还可借助 LIMIT 子句来限制任意查询的输出内容。

示例 17.14

```
1  mysql> SELECT * FROM teams ORDER BY name;
   +----------------+------+--------+
   | name           | wins | losses |
   +----------------+------+--------+
   | Bangor Rams    |   22 |     24 |
   | Bath Warships  |   32 |      3 |
   | Chico Hardhats |   19 |     25 |
   | Fremont Tigers |   24 |     26 |
   +----------------+------+--------+
   4 rows in set (0.00 sec)

2  mysql> SELECT * FROM teams ORDER BY name DESC;
   +----------------+------+--------+
   | name           | wins | losses |
   +----------------+------+--------+
   | Fremont Tigers |   24 |     26 |
   | Chico Hardhats |   19 |     25 |
   | Bath Warships  |   32 |      3 |
   | Bangor Rams    |   22 |     24 |
   +----------------+------+--------+
   4 rows in set (0.00 sec)

3  mysql> SELECT name, wins FROM teams ORDER BY WINS LIMIT 2;
   +----------------+------+
   | name           | wins |
   +----------------+------+
   | Chico Hardhats |   19 |
   | Bangor Rams    |   22 |
   +----------------+------+
   2 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出所有的字段，并把结果集按名称升序排列。
2. SELECT 语句从名叫 teams 的表中检索出所有的字段，并把结果集按名称降序排列。
3. SELECT 语句检索球队的名称和胜利次数，并按胜利次数升序排列，最后把结果集限定为前两名的球队。

对表进行连接。　如果某个数据库设计得比较好的话，其各个表便会基于某些标准相互关联起来；例如，在上述数据库中，每支球队都有球队名称，而每个教练也都拥有姓名和所在球队的名称。连接（join）操作允许把两个或多个表组合到一起，并基于它们之间的联系返回一个结果集合。有很多种不同的连接（join）语句（内连接、交叉连接、左连接等等），不过它们都遵循基本的 SELECT 语法，只不过需要加上一个 JOIN 子句。

示例 17.15

```
1  mysql> SELECT teams.name, coaches.name, teams.wins
```

```
 FROM teams, coaches WHERE
 teams.name = coaches.team && coaches.id = 4;
+----------------+-----------+------+
| name           | name      | wins |
+----------------+-----------+------+
| Fremont Tigers | Joe Hayes |   24 |
+----------------+-----------+------+
1 row in set (0.00 sec)

2  mysql> SELECT teams.name, coaches.name, teams.wins FROM teams,
    -> coaches  WHERE teams.name = coaches.team &&
-> coaches.title = "Head Coach";
+----------------+-------------+------+
| name           | name        | wins |
+----------------+-------------+------+
| Chico Hardhats | John Doe    |   21 |
| Fremont Tigers | Bud Wilkins |   24 |
+----------------+-------------+------+
2 rows in set (0.00 sec)


3  mysql> SELECT t.name, c.name, t.wins FROM teams t, coaches c
-> WHERE t.name = c.team && c.title LIKE "Head%";
+----------------+-------------+------+
| name           | name        | wins |
+----------------+-------------+------+
| Chico Hardhats | John Doe    |   21 |
| Fremont Tigers | Bud Wilkins |   24 |
+----------------+-------------+------+
2 rows in set (0.00 sec)
```

解释

1. 当球队名称等于教练名字，且教练 ID 为 4 时，SELECT 语句将检索球队名称、教练姓名、以及该球队胜利的次数。在字段前面添加了表名和一个点号，以便标识字段和表。连接（内连接，inner join）意味着所有不匹配的记录都会予以抛弃。只有那些匹配于 WHERE 子句所述标准的行才会显示到结果集中。
2. 当球队名称等于教练名字，且教练头衔是 "Head Coach" 时，SELECT 语句将检索球队名称、教练姓名、以及该球队胜利的次数。与上一个示例相似，连接（内连接，inner join）意味着只有那些匹配于 WHERE 子句所述标准的行才会显示到结果集中。
3. 当球队名称等于教练名字，且教练头衔以“Head”开头时，SELECT 语句将检索球队名称、教练姓名、以及该球队胜利的次数。其中的字母 ‘t’ 和 ‘c’ 又称作这两张表 teams 与 coaches 的别名（alias）。别名机制能节省大量的输入动作。

删除行。 DELETE 命令允许读者从表中移除指定的行。DELETE 和 SELECT 之间惟一真正的区别在于，DELETE 会根据某些标准移除记录，而 SELECT 则负责检索这些记录。

示例 17.16

```
1  mysql> SELECT name FROM teams;
+----------------+
| name           |
+----------------+
```

```
| Fremont Tigers |
| Chico Hardhats |
| Bath Warships  |
| Bangor Rams    |
+----------------+
4 rows in set (0.00 sec)

2  mysql> DELETE FROM teams WHERE name = "Bath Warships";
Query OK, 1 row affected (0.20 sec)

3  mysql> SELECT name FROM teams;
+----------------+
| name           |
+----------------+
| Fremont Tigers
| Chico Hardhats |
| Bangor Rams    |
+----------------+
4 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出 name 字段的所有值。
2. 当球队名称是“Bath Warships”时，DELETE 语句便会移除 name 字段的所有值。
3. SELECT 语句检索出 name 字段的所有值，说明了前面所述的 DELETE 语句已经删去了“Bath Warships”球队。

在表中更新数据。 UPDATE 命令用于对表进行编辑；即，修改或更改某张表中的数据。该语句通过 SET 子句将业已存在的值更改为其他内容，如示例 17.17 所示。

示例 17.17

```
1 mysql> SELECT * FROM teams;
+----------------+------+--------+
| name           | wins | losses |
+----------------+------+--------+
| Fremont Tigers |   24 |     26 |
| Chico Hardhats |   19 |     25 |
| Bath Warships  |   32 |      3 |
| Bangor Rams    |   22 |     24 |
+----------------+------+--------+
4 rows in set (0.00 sec)


1  mysql> UPDATE teams SET wins=wins + 2 WHERE name="Chico Hardhats";
   Query OK, 1 row affected (0.02 sec)
   Rows matched: 1  Changed: 1  Warnings: 0

2  mysql> UPDATE teams SET name="Bath Destroyers"
    ->  where name="Bath Warships";
      Query OK, 1 row affected (0.13 sec)
      Rows matched: 1  Changed: 1  Warnings: 0

3   mysql> SELECT * FROM teams;
+------------------+------+--------+
| name             | wins | losses |
+-------------------+------+--------+
```

```
| Fremont Tigers     |   24 |      26 |
| Chico Hardhats     |   21 |      25 |
| Bath Destroyers    |   32 |       3 |
| Bangor Rams        |   22 |      24 |
+--------------------+------+--------+
4 rows in set (0.00 sec)
```

解释

1. SELECT 语句从名叫 teams 的表中检索出所有的行。在对表进行更新之后，我们将把这个结果集与第 3 行进行对比。
2. UPDATE 语句负责编辑 name 字段。它会把球队“Bath Warships”更名为“Bath Destroyers”。
3. SELECT 语句从名叫 teams 的表中检索出 name 字段的所有值，由此可见 UPDATE 语句已经成功更改了球队“Bath Warships”的名称。

更改（alter）表。ALTER TABLE 命令允许用户添加或删除数据列，从而更改某个已存在的表结构。ALTER 语句拥有多种可选的子句，譬如 CHANGE、MODIFY、RENAME、DROP 等。请不要把 ALTER 与 UPDATE 混为一谈。表的更改（alter）命令能够在创建好一张表之后改变描述该表的结构。用户可以通过它来添加主键、索引、改变某一列的定义或者该列在表中的位置，等等。下面的示例演示了其中一些更改操作。

添加一列

示例 17.18

```
1  mysql> ALTER TABLE teams ADD captain varchar(100);
Query OK, 11 rows affected (0.64 sec)
Records: 11 Duplicates: 0 Warnings:

2  mysql> select * from teams;
+-------------------------+------+--------+---------+
| name                    | wins | losses | captain |
+-------------------------+------+--------+---------+
| Fremont Tigers          |   24 |     26 | NULL    |
| Bath Destroyers         |   34 |      3 | NULL    |
| Chico Hardhats          |   21 |     25 | NULL    |
| Bangor Rams             |   23 |      5 | NULL    |
4 rows in set (0.01 sec)
```

解释

ALTER 语句把一个叫作 captain 的新字段添加到 teams 表中。这个新字段中的值最多可由 100 个字符组成。

删除一列

示例 17.19

```
mysql> ALTER TABLE teams DROP captain;
Query OK, 11 rows affected (0.34 sec)
Records: 11  Duplicates: 0  Warnings: 0
```

添加主键

在示例 17.20 中更改了 teams 表，并将 name 字段设定为一个主键。也就是说，所有新加入的

球队都将拥有惟一的名称。

示例 17.20

```
1  mysql> ALTER TABLE teams MODIFY name VARCHAR(100) NOT NULL,
     -->  ADD PRIMARY KEY(name);
     Query OK, 10 rows affected (0.06 sec)
     Records: 10  Duplicates: 0  Warnings: 0

2  mysql> DESCRIBE teams;
   +--------+------------------+------+-----+---------+-------+
   | Field  | Type             | Null | Key | Default | Extra |
   +--------+------------------+------+-----+---------+-------+
   | name   | varchar(100)     | NO   | PRI | NULL    |       |
   | wins   | int(10) unsigned | YES  |     | NULL    |       |
   | losses | int(10) unsigned | YES  |     | NULL    |       |
   +--------+------------------+------+-----+---------+-------+
   3 rows in set (0.05 sec)
```

删除表。 表的删除（drop）操作相对较为简单。只需使用 drop 命令和表名即可。

示例 17.21

```
mysql> DROP TABLE teams;
Query OK, 5 rows affected (0.11 sec)
```

删除数据库。 如需删除某个数据库，可使用 drop database 命令。

示例 17.22

```
mysql> DROP DATABASE sample_db;
Query OK, 1 row affected (0.45 sec)
```

17.4 什么是 Perl DBI

DBI 是一层位于应用程序与一到多个数据库驱动模块之间的“胶水”。

—— Tim Bunce，DBI 的作者

DBI 的全称是数据库独立接口（Database Independent Interface）。DBI 是一个面向对象的模块，允许用户的 Perl 程序使用相同的方法调用、变量和规范来访问许多不同类型的数据库。它会为指定的数据库系统定位数据库驱动模块（DBD），并能动态地加载合适的 DBD 模块。这些数据库驱动含有访问指定数据库时所必需的库。例如，为了连接一个 MySQL 数据库，用户必须安装 DBD-MySQL 驱动; 而为了访问一个 Oracle 数据库，则需要 DBD-Oracle 驱动。DBI 在 Perl 脚本与数据库驱动模块之间扮演着接口（interface）的角色；也就是说，它会将 Perl 的输出内容转译为某种特定驱动能够理解的代码，而不管该驱动是 Oracle，Sybase 还是 MySQL，等等。用户只需建立 SQL 查询字符串，并通过一个 DBI 方法将它发送到合适的数据库驱动，然后便能得到返回结果。不论使用的是什么数据库，用户都能在 Perl 程序中以相同的方式管理它们。如图 17-19 所示。

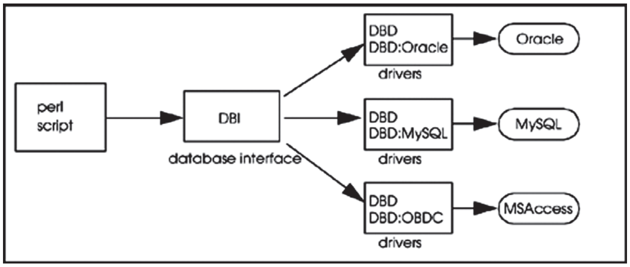


图 17-19 DBI 与驱动

17.4.1 安装 DBI

通过 PPM 安装 DBI-MySQL。 如果用户正在使用来自 ActiveState 的 PPM（Perl 包管理器）的话，安装 DBI 就是一件非常容易的事情。PPM 是一种包管理实用程序，它能简化 Perl 模块的搜索、安装、更新和移除工作。当前默认的 PPM 版本都提供了图形化用户界面，不过读者也能使用示例 17.23 所示的命令行界面。

PPM 快速入门

下面的示例摘自 PPM 初学者指南，它们能帮助用户学习如何在命令行中通过 PPM 按部就班地安装、移除来自 CPAN 的模块。

示例 17.23

```
(At the command line)
ppm help quickstart
quickstart -- a beginners' guide to PPM3
Description
  PPM (Programmer's Package Manager) is a utility for managing
  software "packages". A package is a modular extension for a
  language or a software program. Packages reside in repositories.
  PPM can use three types of repositories:

   1) A directory on a CD-ROM or hard drive in your computer
   2) A website
   3) A remote Repository Server (such as ASPN)

  Common Commands:
  To view PPM help:
    help
    help <command>

 To view the name of the current repository:
     repository

 To search the current repository:
     search <keywords>

 To install a package:
     install <package_name>

  Most commands can be truncated; as long as the command is
  unambiguous, PPM will recognize it. For example, 'repository add
```

```
  foo' can be entered as 'rep add foo'.

  PPM features user profiles, which store information about
  installed packages. Profiles are stored as part of your ASPN
  account; thus, you can easily maintain package profiles for
  different languages, or configure one machine with your favorite
  packages, and then copy that installation to another machine by
  accessing your ASPN profile.

  For more information, type 'help profile' at the PPM prompt.
```

利用 PPM 的安装步骤。 这里列出了安装所需模块的简单步骤。

1. 请确认已经连接上了 Internet。
2. 在 shell 命令行提示符下（MS-DOS 或 UNIX）键入：ppm。
3. 这时会显示一个 PPM 提示符：ppm>。
4. 如需查看选项，可在 PPM 提示符下输入：help。
5. 如需查看可用模块，首先可以搜索一个需要的模块，然后使用其 install 命令。

示例 17.24

```
1 $ ppm
 PPM - Programmer's Package Manager version 3.1.
 Copyright (c) 2001 ActiveState Corp. All Rights Reserved.
 ActiveState is a devision of Sophos.

Entering interactive shell. Using Term::ReadLine::Stub as readline
library.

Type 'help' to get started.

Setting 'target' set to 'ActivePerl 5.8.4.810'.
ppm>

2 ppm> query *
Querying target 1 (ActivePerl 5.8.7.813)
   1. ActivePerl-DocTools         [0.04] Perl extension for
Documentation TOC Generation
   2. ActiveState-RelocateTree    [0.03] Relocate a Perl installation
   3. ActiveState-Rx              [0.60] Regular Expression Debugger
   4. Archive-Tar                 [1.07] Manipulates TAR archives
   5. Compress-Zlib              [1.22] Interface to zlib compression
library
   6. Data-Dump                   [1.01] Pretty printing of data
structures
   7. DBD-mysql               [2.9003] MySQL driver for the Perl5
                          Database Interface (DBI)
   8. DBI             [1.47] Database independent interface for Perl
   9. Digest                     [1.0] Modules that calculate
message digests
  10. Digest-HMAC               [1.01] Keyed-Hashing for Message
Authentication
  11. Digest-MD2                [2.03] Perl interface to the MD2
Algorithm
  12. Digest-MD4                 [1.1] Perl interface to the MD4
```

```
Algorithm
  13. Digest-MD5          [2.20] Perl interface to the MD5 Algorithm
  14. Digest-SHA1               [2.06] Perl interface to the SHA-1
Algorithm
  15. File-CounterFile          [1.01] Persistent counter class
  16. Font-AFM                  [1.18] Interface to Adobe Font
Metrics files
  17. HTML-Parser               [3.34] HTML parser class
  18. HTML-Tagset              [3.03] Data tables useful in parsing
HTML
  19. HTML-Tree                [3.18] build and scan parse-trees of
HTML
  20. IO-Zlib                   [1.01] IO:: style interface to
Compress::Zlib
  21. libwin32                      [0.21] A collection of extensions
that aims to provide comp~
  22. libwww-perl                   [5.75] Library for WWW access in
Perl
  23. Mail-Sendmail                 [0.79] Simple platform independent
mailer
  24. MD5                           [2.02] Perl interface to the MD5
Algorithm (obsolete)
  25. MIME-Base64                   [2.12] Encoding and decoding of
base64 strings
  26. PPM                         [2.1.6] Perl Package Manager: locate,
install, upgrade softw~
  27. PPM-Agent-Perl               [3.0.4] PPM Installer Backend for
Perl
  28. PPM3                          [3.1] Perl Package Manager: locate,
install, upgrade softw~
  29. SOAP-Lite                     [0.55] Library for Simple Object
Access Protocol (SOAP) cli~
  30. Storable                    [1.0.12] persistency for perl data
structures
  31. Tk                         [800.024] A Graphical User Interface
Toolkit
  32. URI                           [1.27] Uniform Resource Identifiers
(absolute and relative)
  33. Win32-AuthenticateUser       [0.02] Win32 User authentication for
domains
  34. XML-Parser                   [2.34] A Perl module for parsing XML
documents
  35. XML-Simple                    [2.09] Easy API to read/write XML
(esp config files)
ppm>

3  ppm> describe DBI
   ====================
       Name: DBI
    Version: 1.50
     Author: Tim Bunce (dbi-users@perl.org)
      Title: DBI
   Abstract: Database independent interface for Perl
   Location: ActiveState PPM2 Repository
Available Platforms:
       1. MSWin32-x86-multi-thread-5.8
====================
```

```
4  ppm> describe DBD-mysql
    ====================
    Name: DBD-mysql
 Version: 3.0002
  Author: Patrick Galbraith (patg@mysql.com)
   Title: DBD-mysql
Abstract: A MySQL driver for the Perl5 Database Interface (DBI)
Location: ActiveState PPM2 Repository
Prerequisites:
       1. DBI 0.0
Available Platforms:
       1. MSWin32-x86-multi-thread-5.8
====================
ppm>q

To install the DBI (Database Interface for Perl) and DBD-mysql (Driver
for MySQL) from CPAN, start PPM and execute the install command as
follows:


5      ppm> query *
6      ppm> install DBI
7      ppm> install DBD-mysql
```

PPM GUI。 在刚刚显示 Perl 包管理器时，它会从数据库中同步其 ActiveState 知识库（repository）。用户可以从中看到所有已经安装的包以及知识库中全部的包，并可使用搜索框来寻找所要的模块。如果找到的话，则高亮显示该模块，如图 17-20 所示。

如要安装标记的包，用户可以点击位于屏幕上方右侧部分的绿色箭头。

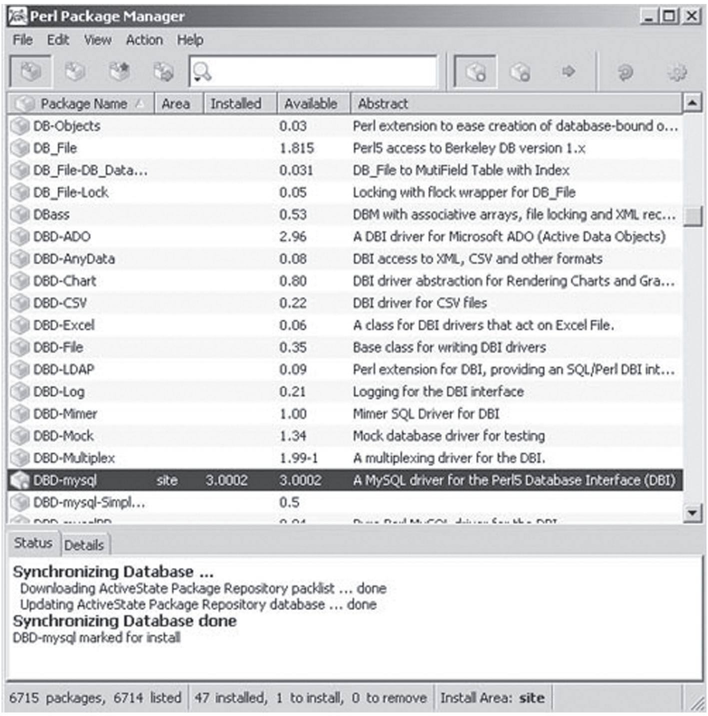


图 17-20 标记待安装的包

在 Linux 中使用 PPM。 如果正在使用 Linux 的话，用户可以以 rpm（Redhat 包管理器）文件或 tar 文件的形式从 ActiveState 中下载 Perl 5.8.8。有关下载 ActivePerl 的指南文档可从 http://aspn.activestate.com/ASPN/docs/ActivePerl/5.8/install.html 处找到。然后，在 Perl 目录中运行 shell 脚本 install.sh。执行完安装脚本后，请设置 PATH，使它指向 Perl 目录。然后便可像在 Windows 中那样使用 PPM 程序安装来自 CPAN 的模块了。上述过程避免了使用 CPAN 时可能碰到的诸多问题和麻烦。

为了安装所需模块，读者必须连接到 Internet 上。如需查看已经安装的模块，请键入：

```
\rpm -i Active-State.....rpm
sh install.sh
```

使用 CPAN 安装 DBI。 用于维护本地 Perl 发布包的主要工具是 CPAN 模块，该模块能够访问 Perl 综合文献网络（Comprehensive Perl Archive Network），又称 CPAN。请在命令行提示符下键入：

```
$ Perl -MCPAN -e shell
```

在 CPAN 提示符下键入“help”，便可看到一组选项。如需获得最新版本的 CPAN，请在 CPAN 提示符后面键入：

```
cpan > install Bundle::CPAN
```

在回答完连珠炮似的诸多问题后，如果看到下面这段提示信息，请读者不必惊讶。

Stop.
nmake -- NOT OK
Running make test
Can't test without successful make
Running make install
make had returned bad status, install seems impossible

由此可以看到，为什么使用 PPM 会更容易一些。

在 Redhat 中，请尝试如下步骤：

1. 安装最新版本的 MySQL 服务器 / 客户端 RPM 包（RedHat 自带的 MySQL 往往不能很好地工作）。MySQL-client-standard-5.0.24-0.rhel3.i386.rpm。

2. 安装 Perl-DBI 和 MySQL-devel 模块（RedHat Enterprise Linux ES 版本 3（Taroon）中已经带有这两个 RPM 包。）

Perl-DBI-1.32-5.i386.rpm

mysql-devel-3.23.58-l.i386.rpm

3. 执行下列命令，从 CPAN 中获取 DBD-MySQL：

```
Perl -MCPAN -e ' install Bundle::DBD::mysql '
```

示例 17.25

```
cpan> h

Display Information
 command  argument          description
 a,b,d,m  WORD or /REGEXP/   about authors, bundles, distributions,
modules
```

```
 i        WORD or /REGEXP/   about anything of above
 r        NONE               reinstall recommendations
 ls       AUTHOR             about files in the author's directory

Download, Test, Make, Install...
 get                        download
 make                       make (implies get)
 test      MODULES,         make test (implies make)
 install   DISTS, BUNDLES   make install (implies test)
 clean                      make clean
 look                       open subshell in these dists' directories
 readme                     display these dists' README files

Other
 h,?           display this menu       ! perl-code    eval a perl
command
 o conf [opt]  set and query options   q              quit the cpan
shell
 reload cpan   load CPAN.pm again      reload index   load newer
indices
 autobundle    Snapshot                force cmd      unconditionally
do cmd
cpan>
```

17.4.2 DBI 类方法

DBI 模块是面向对象的，它带有很多方法和变量。有关该模块的文档将在后面列出。数据库对象又称作句柄（handle）。数据库句柄（database handle）能连接到指定的数据库上，而语句句柄（statement handle）则用于向数据库发送 SQL 语句。请注意，像 $dbi、$sth、$rc 这样的名称可用于描述语句句柄、返回代码、数据行，等等。（这些名称是文档中的规范；例如，$dbh 表示一个数据库句柄，而 $sth 则表示一个语句句柄。）

```
$ perldoc DBI
 Notation and Conventions
   The following conventions are used in this document:

     $dbh    Database handle object
     $sth    Statement handle object
     $drh    Driver handle object (rarely seen or used in applications)
     $h      Any of the handle types above ($dbh, $sth, or $drh)
     $rc     General Return Code  (boolean: true=ok, false=error)
     $rv     General Return Value (typically an integer)
     @ary    List of values returned from the database, typically a row
of dat

     $rows   Number of rows processed (if available, else -1)
     $fh     A filehandle
     undef   NULL values are represented by undefined values in Perl
     \%attr  Reference to a hash of attribute values passed to methods

   Note that Perl will automatically destroy database and statement
   handle objects if all references to them are deleted.

NAME
```

```
    DBI - Database independent interface for Perl

SYNOPSIS
      use DBI;
      @driver_names = DBI->available_drivers;
      @data_sources = DBI->data_sources($driver_name, \%attr);
      $dbh = DBI->connect($data_source, $username, $auth, \%attr);

      $rv  = $dbh->do($statement);
      $rv  = $dbh->do($statement, \%attr);
      $rv  = $dbh->do($statement, \%attr, @bind_values);

      $ary_ref  = $dbh->selectall_arrayref($statement);
      $hash_ref = $dbh->selectall_hashref($statement, $key_field);

      $ary_ref  = $dbh->selectcol_arrayref($statement);
      $ary_ref  = $dbh->selectcol_arrayref($statement, \%attr);

      @row_ary  = $dbh->selectrow_array($statement);
      $ary_ref  = $dbh->selectrow_arrayref($statement);
      $hash_ref = $dbh->selectrow_hashref($statement);

      $sth = $dbh->prepare($statement);
      $sth = $dbh->prepare_cached($statement);

      $rc = $sth->bind_param($p_num, $bind_value);
      $rc = $sth->bind_param($p_num, $bind_value, $bind_type);
      $rc = $sth->bind_param($p_num, $bind_value, \%attr);
      $rv = $sth->execute;
      $rv = $sth->execute(@bind_values);
      $rv = $sth->execute_array(\%attr, ...);

      $rc = $sth->bind_col($col_num, \$col_variable);
      $rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);

      @row_ary  = $sth->fetchrow_array;
      $ary_ref  = $sth->fetchrow_arrayref;
      $hash_ref = $sth->fetchrow_hashref;

      $ary_ref  = $sth->fetchall_arrayref;
      $ary_ref  = $sth->fetchall_arrayref( $slice, $max_rows );

      $hash_ref = $sth->fetchall_hashref( $key_field );

      $rv  = $sth->rows;

      $rc  = $dbh->begin_work;
      $rc  = $dbh->commit;
      $rc  = $dbh->rollback;

      $quoted_string = $dbh->quote($string);

      $rc  = $h->err;
      $str = $h->errstr;
      $rv  = $h->state;
```

```
$rc  = $dbh->disconnect;

*The synopsis above only lists the major methods and parameters.*
```

17.4.3 如何使用 DBI

在通过 use DBI 语句将 DBI 模块加载到程序中之后，还需执行如下五个步骤：连接到数据库、准备一条查询、执行该查询、获取结果、以及断开连接。

为连接到 MySQL 数据库，请使用 connect() 方法。该方法会指定数据库类型（MySQL、Oracle、Sybase、CSV 文件、Informix 等）、数据库名、主机名、用户及其口令、以及一些附加的用于指定错误处理与事务处理方式的可选参数等（手册页中用到了 $dbh，但实际上用户可通过任意的标量名来调用）。

在连接到 MySQL 数据库之后，用户就持有了一个数据库句柄（引用了相应的数据库对象）。这时，用户便可通过准备并执行一条 SQL 语句的方式来发送查询。为此可以调用 prepare() 与 execute() 方法，亦可使用 do() 方法。prepare() 与 execute() 方法适用于 SELECT 语句，而 do() 方法则一般用于不返回任何结果集的 SQL 语句，譬如 INSERT、UPDATE 或 DELETE 语句。这些方法返回的结果取决于各自查询的返回值。例如，成功执行的 SELECT 查询语句将会返回一个结果集（在 DBI 手册页中表示为 $sth)；通过 do() 方法成功执行的 INSERT/UPDATE/DELETE 查询语句会返回影响到的行数；而执行失败的查询则会返回一条错误或 undef。上述大部分数据都是以字符串的形式返回给 Perl 的，而 null 值则会返回为 undef。

当把查询发送至数据库并返回结果集（指向结果对象的引用）之后，便可借助一些特殊方法来解析这些数据，譬如 fetchrow_array() 和 fetchrow_hashref()。这些方法会分别将每一条记录当作 Perl 数组或 Perl 散列来处理。

最后，当完成上述操作后，可使用 finish() 方法释放由 prepare() 方法返回的结果对象，然后通过 disconnect() 方法断开数据库并终止此次会话。

现在我们将详细介绍上述五个步骤。

17.4.4 连接和断开数据库

当载入完成后，DBI 模块就负责为给定的数据库加载合适的驱动。然后用户便可使用该模块提供的方法操纵数据库了，所有这些方法都可在 perldoc DBI 命令的输出内容中看到。我们将用到的第一个方法是 connect() 方法，它负责获取数据库连接；用到的最后一个方法则是 disconnect()，它负责断开连接。

检查可用的 DBI 数据库驱动

示例 17.26

```
use DBI;
my @drivers = DBI->available_drivers;
print join(", ", @drivers),"\n";

(Output for Windows)
DBM, ExampleP, File, Gofer, Proxy, SQLite, Sponge, mysql
```

connect() 方法。 connect() 方法负责向指定数据库发起一个连接，并返回一个称作数据库句柄（database handle）的对象。通过使用多个 connect 语句，用户可以在一个程序中对同一数据库发起多个连接，甚至还可同时连接到不同的数据库上。connect() 方法允许接受如下几个参数：

```
"dbi:$driver:$database, $port, $username, $password"
```

1. 第一个参数是 DSN（数据源名称）字符串，它是数据库的逻辑名称。任何定义数据源的属性都能赋值给 DSN，并由驱动负责检索。DSN 含有 DBI 模块名称，其格式是：首先是 DBI，后面跟有一个冒号，其后是数据库驱动（MySQL、Sybase、Oracle），接着是另一个冒号，以及待连接数据库的实际名称和 / 或主机名（默认是“localhost”）、端口等，最后以一个分号结尾。
2. 下一个连接参数是用户名。
3. 然后是该用户的口令（除非强制要求，否则它是可选的）。
4. 最后是一个指向散列的引用（该集合含有用于错误处理、自动提交等工作的可选属性）。

示例 17.27

```
1 $dbh=
DBI->connect("dbi:<RDMS>:<database>","<username>","<password>",
                 \%attributes) or die("Couldn't connect");

2 $dbh=DBI->connect('DBI:mysql:sample_db','root','quigley1') or
                                       die "Can't connect";

3 $dbh=DBI->connect('DBI:mysql:database=sample_db;user=root;
                                       password=quigley1');

4 $dsn = dbi:mysql:northwind; $username="root"; $password="letmein";
  $dbh = DBI->connect($dsn, $user, $password,
      { PrintError => 0, RaiseError => 1, AutoCommit => 0 });

-------------------Using Other Database Systems------------

5 $dbh = DBI->connect('dbi:Oracle:payroll','scott','tiger');
  $dbh = DBI->connect("dbi:Oracle:host=torch.cs.dal.ca;sid=TRCH",
$user, $passwd);
  (Oracle)

6 $dbh = DBI->connect('dbi:odbc:MSS_pubs','sa', '12mw_1');
  (MS SQL Server)
```

解释

1. connect 方法将返回一个数据库句柄。本行展示了连接数据库的调用格式。用户至少必须提供 DSN 字符串，它定义了模块的名称，包括 dbi、数据库驱动名、以及表示为 <database> 的数据库名。用户名、主机名、口令和其他属性都是可选的。
2. connect 方法的参数必须位于同一行内，并且参数周围不能出现空格。
3. DBI->errstr 返回连接失败的原因——例如，“Bad Password”。
4. 本次连接将通过用户名“root”和口令“letmein”连接到一个名叫“northwind”的 MySQL 数据库上。
5. 本次连接将通过用户名“scott”和口令“tiger”连接到一个名叫“payroll”的 Oracle 数据库上。
6. 名叫“MSS_pubs”的数据源用到了用户名“sa”与口令“12mw_1”，还有 ODBC 驱动。

17.4.5 disconnect() 方法

读者或许还记得第 10 章中介绍的内容，当我们通过用户自定义的文件句柄打开某个文件之后，必须在结束时通过内建的 close 函数来关闭它；如果没有关闭它的话，最后则会在 Perl 脚本退出时由操作系统负责关闭。关闭数据库也与之类似。当用户不再使用数据库之后，最好能通过 disconnect 方法来关闭这个数据库连接。当然，由于数据库句柄只是一个对象，所以当程序退出或对象离开作用域时，Perl 将会自动移除指向该对象的所有引用。

```
$dbh->disconnect();
```

17.4.6 准备语句句柄并获取结果

在查询数据库时，SQL 的 select 语句可能是最为常用的语句。在为数据库准备 select 语句时，会将查询作为一个字符串参数发送给 DBI 的 prepare 方法。该查询语句和在 MySQL 控制台中输入的查询几乎一模一样，只不过前者没有分号。数据库句柄会调用 prepare 方法。数据库负责决定查询的处理方法（创建“计划”），DBI 则负责返回一个语句句柄，其中含有有关如何执行该查询的细节信息。语句句柄（表示为 $sth）会把将在数据库中执行的独立的 SQL 语句封装到一起，并调用 execute 方法。

execute 方法会让数据库执行 SQL 语句（执行既有的“计划”），并将结果集返回给程序。如果发生错误的话，便会返回一个 undef。不论影响到多少行内容，一次成功的执行总会返回真（true），甚至在只影响零行内容时也是如此。一旦执行完毕，便会丢弃这个“计划”（详见 prepare_cache 方法）。创建并执行的语句句柄数量基本上是不受限制的。

即使在执行完这些计划后，用户也无法直接看到结果集，除非使用其他一些 DBI 方法检索它，譬如 dump_results()、fetchrow_array() 或 fetch() 等。

选择、执行并转录（dump）结果。 DBI 的 dump_results 方法能从语句句柄对象中获取所有的行，并在一个简单的语句中打印这些结果。

连接、准备、执行并转录（dump）结果

示例 17.28

```
    (The Script)
    use DBI;
1   $db=
DBI->connect('DBI:mysql:sample_db;user=root;password=quigley1');

2   $sth=$db->prepare("SELECT * FROM coaches")
    or die "Can't prepare sql statement" . DBI->errstr;
3   $sth->execute();
    print qq(\n\tContents of "coaches" table\n);
4    $sth->dump_results();    # Display results of the execute

5    $sth->finish();
6    $dbh->disconnect();

(Output)
         Contents of "coaches" table
'1', 'John Doe', 'Chico Hardhats', 'Head Coach', '2002-12-10'
```

```
'2', 'Jack Mattsone', 'CHardhats', 'Offensive Coach', '2004-10-05'
'3', 'Bud Wilkins', 'Fremont Tigers', 'Head Coach', '1999-09-06'
'4', 'Joe Hayes', 'Fremont Tigers', 'Defensive Coach', '1998-06-16'
'5', 'George Jones', 'Bangor Rams', 'Offensive Coach', '2003-09-03'
'6', 'Jerry O'Connell','Portland Penguins','Head Coach', '2006-02-22'
6 rows
```

解释

1. 向名为 sample_db 的 MySQL 数据库发起连接,并返回一个名叫 $dbh 的数据库句柄。它是一个表示该连接的对象。现在就可以访问数据库了。
2. prepare 方法用于准备 SQL 查询。它会返回一个语句句柄，即一个负责封装并执行查询的对象。（请注意，这里的 SQL 语句并不以分号结尾。）
3. execute 方法使得查询真正得以执行。这时便可检索查询的结果。
4. 语句句柄调用 dump_results 方法，并打印查询的结果。

选择、执行并将一行获取为数组。 当调用 fetchrow_array() 方法时，数据库将会把结果的第一行返回为一个数组，其中每个字段都是数组中的一个元素。每次成功调用 fetchrow_array() 都会把当前位置移动到结果的下一行上，直到没有更多的结果可供获取，这时才会生成一个 undef 值。如示例 17.29 所示，可使用一个 while 或 for 循环来获取所有的行。（值得注意的一点是，当获取各个字段时，其排列顺序与 SQL 查询中列出的顺序是相同的。）

连接、准备、执行并通过 fetchrow_array() 获取数据

示例 17.29

```
(The Script)
    use DBI;
1   my $dbh=DBI->connect(qq(DBI:mysql:database=sample_db;user=root;
         password=quigley1)) or die "Can't connect";
2   my $sth=$dbh->prepare("SELECT name, wins, losses FROM teams");

3   $sth->execute();
    print "Contents of sample_db, the mysql database.\n\n";

4    while(my @row=$sth->fetchrow_array()){ # Get one row at a time
5        print "name=$row[0]\n";       # Field one
         print "wins=$row[1]\n";       # Field two
         print "losses=$row[2]\n\n";   # Field three
           }
6   print $sth->rows, " rows were retrieved.\n";

7   $sth->finish();
8   $dbh->disconnect();

(The Output)
Contents of sample_db, the mysql database.

name=Fremont Tigers
wins=24
losses=26

name=Chico Hardhats
```

```
wins=19
losses=25

name=Bath Warships
wins=32
losses=3

name=Bangor Rams
wins=22
losses=24

4 rows were retrieved.
```

解释

1. connect() 方法会返回一个数据库句柄 $dbh，它是一个对象，引用了名叫 sample_db 的 MySQL 数据库。
2. 准备一个 SQL select 语句，返回一个名为 $sth 的语句句柄。
3. 将查询发送到数据库并执行之。
4. fetchrow_array() 方法返回来自数据库的第一行内容，其中的各个字段分别对应于数组 @row 的元素。如需获取后面各行内容，可使用一个 while 循环。当没有更多行可供处理时，循环即告一段落。
5. 一行中的每个字段都会赋值给一个变量，并打印出来。
6. rows() 方法会返回语句句柄影响到的行数。
7. finish 方法会释放语句句柄。
8. disconnect 方法负责释放数据库句柄。

选择、执行并将一行获取为散列。 fetchrow_hashref() 方法会把来自数据库表的一行内容获取为一个散列引用，其键是列名，值则是该列所保存的数据。下面这个示例和前面的示例几乎一样，所不同的是它用 fetchrow_hashref() 方法代替了 fetchrow_array()。

示例 17.30

```
   use DBI;
1  $dbh=DBI->connect(qq(DBI:mysql:database=sample_db;user=root;
          password=quigley1)) or die "Can't connect";

2  $sth=$dbh->prepare("SELECT name, wins, losses FROM teams") ;

3  $sth->execute();
   $count=0;

   print "Contents of sample_db, the mysql database.\n\n";

4  while(  my $row = $sth->fetchrow_hashref()){
5     print "Name:     $row->{name}\n";
    print "Wins:     $row->{wins}\n";
    print "Losses:   $row->{losses}\n\n";
    $count++;
  }

6  print "There are $count rows in the sample database.\n";
```

```
7  $sth->finish();
8  $dbh->disconnect();
```

解释

1. connect() 方法会返回一个数据库句柄 $dbh，它是一个对象，引用了名叫 sample_db 的 MySQL 数据库。
2. 准备一个 SQL select 语句，返回一个名为 $sth 的语句句柄。
3. 将查询发送到数据库并执行之。
4. fetchrow_hashref() 方法把来自数据库的第一行内容返回为指向一个匿名散列的引用，该匿名散列由多个键 / 值对组成。它的键是表内字段的名称，值则是字段保存的内容。如需获取后面各行内容，可使用一个 while 循环。当没有更多行可供处理时，循环即告一段落。
5. 打印键（key）所指定的字段的所有值。
6. finish 方法负责释放语句句柄。
7. disconnect 方法负责释放数据库句柄。

17.4.7 处理引号

在把字符串发送到数据库时，必须把它们置于引号内。字符串本身也可能含有引号，譬如字符串 "Mrs. O'Donnell"，而这些引号必须在发送到数据库时以适当的方式予以舍弃（escape）。更为复杂的是，不同的数据库系统处理引号的规则也各不相同。DBI 通过其提供的 quote 方法来处理引用问题。该方法必须与某个数据库句柄配合使用，它会根据指定数据库所定义的规则，将一个字符串转换为正确舍弃后的字符串形式。

示例 17.31

```
   use DBI;
   $dbh=DBI->connect(qq(DBI:mysql:database=sample_db;user=root;
          password=quigley1)) or die "Can't connect";
1  $namestring=qq(Jerry O'Connell);
2  $namestring=$dbi->quote($string);
3  print $namestring;
4  $sth=$dbi->prepare("SELECT * FROM coaches WHERE name=$namestring")
   or die "Can't prepare sql statement" . DBI->errstr;
   $sth->execute();
5  print qq(\nContents of "coaches" table\n);

6        while(my @val = $sth->fetchrow_array()){
           print "\tid=$val[0]\n";
           print "\tname=$val[1]\n";
           print "\tteam_name=$val[2]\n";
           print "\tteam_name=$val[3]\n";
           print "\tstart_date=$val[4]\n\n";
         }

    $sth->finish();
    $dbh->disconnect();

     (Output)
2   'Jerry O\'Connell'
5   Contents of "coaches" table
```

```
6         id=6
          name=Jerry O'Connell
          team_name=Portland Penguins
          team_name=Head Coach
          start_date=2006-02-22
```

解释

1. 将字符串变量 $namestring 赋值为一个含有单引号的字符串。
2. 使用一个 DBI quote 方法来为 mysql 数据库准备字符串，它会把字符串放入引号之间，并在那个单引号前面加上反斜杠。
3. 本行向用户展示了 quote 方法是如何准备字符串的。请注意，O'Connell 中的省略符已通过反斜杠予以屏蔽。
4. 在 WHERE 子句中，上述正确引用的字符串将用于判断它的值是否与 coaches 表中的某个教练姓名相匹配。下面这一行展示了 MySQL 客户端是如何插入数据的。读者可以看到它是如何在字符串中管理引号的；即，把单引号嵌入到双引号中：

   mysql> insert into coaches values(',',"Jerry O'Connell",'Portland Penguins', 'Head Coach','2006-2-22');

5 ～ 6. 显示该表内容。

17.4.8 获取错误消息

在使用 DBI 时，了解出错的原因是非常重要的。连接失败了吗？是否正确地准备了 SQL 语句？DBI 定义了多种用于处理错误的方法。用户既可以针对某个指定句柄通过 PrintError 和 RaiseError 属性来使用自动错误处理机制；亦可使用诊断方法和特殊 DBI 变量。

自动错误处理。 当连接到数据库时，DBI 模块提供了自动错误处理机制。用户既可以在每次 DBI 方法失败时获得警告信息，亦可设定编译指示符，以便发送消息并退出程序。在使用 connect 方法时，最为常见的两个属性是 PrintError 和 RaiseError。

PrintError 属性

在默认情况下，connect() 方法会把 PrintError 属性设置为“on”（即设置为 1），并在任意 DBI 方法失败时自动生成一条警告信息。

RaiseError 属性

RaiseError 属性可用于强制生成错误并提供异常。在默认情况下它是关闭的。如果将它设置为“on”，则任何引发错误的 DBI 方法都将造成 DBI 输出错误信息（即 $DBI::errstr）并退出程序。如果把 RaiseError 属性设置为 on 的话，一般都会同时将 PrintError 属性设置为 off。如果 PrintError 属性也为 on 的话，则会优先处理 PrintError 属性。通常会在使用 RaseError 时用到一个 eval 代码块，以便捕获之前抛出的异常。如果在 eval 块中又出现“die”语句、编译错误或运行时错误的话，特殊变量 $@ 就会赋值为错误消息的内容；如果没有出错，则赋值为 null。一般设置了 $@，用户不必退出 DBI 就能处理错误（详见示例 17.32）。

手动错误处理。如果用户希望在某个特定方法失败时手动检查错误，可以使用 DBI 模块提供的错误诊断方法（error diagnostic method）或错误诊断变量（error diagnostic variable）。它让用户能够控制自己调用的每一个方法，以便出错时跟踪其错误信息。

错误诊断方法

首先我们将看到两个错误诊断方法 err() 和 errstr()。这两个方法在调用时可以接受任何合法的句柄、驱动、数据库或语句。err() 方法会返回与碰到的问题相关的出错代码。出错代码是一个数

字，随着所用数据库系统的不同而不同。errstr() 方法对应于上述错误代码数值，不过它返回的是一个含有信息的字符串，能描述为什么上一个 DBI 方法会调用失败。在下一次方法调用之前，对应于某个句柄的所有错误消息都将会重新设置，因此用户必须在该句柄刚刚产生错误时及时地检查错误信息。

```
$rv = $h->err();
$str = $h->errstr();
```

错误诊断变量

DBI 变量 $DBI::err 和 $DBI::errstr 是类变量，其功能类似于前面所述的两个方法，所不同的是它们的声明周期更短，一般只用于上一次调用的方法。$DBI::err 含有对应于上一次调用的方法的错误数值；而 $DBI::errstr 则含有一个字符串，负责描述对应于 $DBI::err 中错误数值的错误消息。一般而言，用户应当及时检查连接操作的返回状态，如果 connect() 方法失败的话，则应打印 $DBI::errstr。

示例 17.32

```
   (The Script)
     use DBI;
     $driver="DBI:mysql";
     $database="sample_db";
     $user="root";
     $host="localhost";

     $dbh=DBI->connect('dbi:mysql:sample_db','root','quigley1',
                     {
1                           RaiseError => 1, # Die if there are errors
2                           PrintError => 0, # Warn if there are errors
                     }
3                 ) or die $DBI::errstr; # Report why connect failed

4    $sth=$dbh->prepare("SELECT name, wins, losses FROM teams") or die
"Can't prepare sql statement" . DBI->errstr;
     $sth->execute();
     print "Contents of sample_db, the mysql database.\n\n";
     while(my @val = $sth->fetchrow_array()){
           print "name=$val[0]\n";
           print "wins=$val[1]\n";
           print "losses=$val[2]\n\n";
                 }
5    print $sth->rows," rows were retrieved.\n";

     $sth->finish();
     $dbh->disconnect();
```

解释

1. 这里我们开启了 RaiseError 属性，如果出现任何由 DBI 方法调用引起的错误，它都将让程序退出。
2. PrintError 属性默认是打开的。如果某个方法失败的话，它就会发送一条错误消息。这里将它设置为 0，即关闭状态，因为 RaiseError 已经打开了。当然，用户亦可把 RaiseError 和 PrintError 同时设置为打开或关闭状态。如果二者都打开的话，首先会由 PrintError 发送警

告信息，然后再由 RaiseError 打印消息并退出程序。

3. 如果连接失败的话，$DBI::errstr 变量将打印出该连接失败的原因。
4. 如果 prepare 方法失败的话，这里会使用 errstr 方法报告可能碰到的错误; 例如，没有正确地准备好 SQL 语句。

错误消息示例

示例 17.33

```
1 (Bad Database Name; connect failed)
    DBI connect('ample_db','root',...) failed: Unknown database
'ample_db'atfirst.dbi line 9

2  (Bad Password; connect failed)
   DBI connect('sample_db','root',...) failed: Access denied for user
'root'@'localhost' (using password: YES) at first.dbi line 9

3  (Bad SQL Query; execute failed)
DBD::mysql::st execute failed: Unknown column 'win' in 'field list' at
first.dbi line 23.
```

绑定列并获取值。 绑定列（binding columns）是最为高效的值获取途径。绑定操作允许用户把某个 Perl 变量关联到数据库中的一个字段（列）值上。当获取该值时，其对应的变量将自动更新为检索得到的值，从而加快获取值的速度。DBI 提供了 bind_columns() 方法，用于将每个列绑定到一个标量型引用上。当调用 fetch() 方法时，就会将来自数据库的值赋值给同名的标量，而不是给前面示例中涉及的数组或散列。

每当调用 fetch() 方法时，这些标量都会更新为来自当前行的值。

（有关另一种绑定列的方式，请查阅 DBI 文档中的 bind_col 相关介绍。）

绑定列

示例 17.34

```
     use DBI;
     my $driver="DBI:mysql";
     my $database="sample_db";
     my $user="root";
     my $host="localhost";

     my $dbh =
 DBI->connect("$driver:database=$database;host=$host;user=$user")
        or die "Can't connect: " . DBI->errstr;

1   my $sth=$dbh->prepare("SELECT name, wins, losses FROM teams")
        or die "Can't prepare sql statement" . DBI->errstr;

2   $sth->execute() or die "Can't prepare sql
        statement". $sth->errstr;
;

3   my($name, $wins, $losses);# Scalars that will be bound to columns
4   $sth->bind_columns(\$name,\$wins,\$losses);
```

```
            # scalar references
        print "\nSelected data for teams.\n\n";
        printf"\t%-20s%-8s%-8s\n","Name","Wins", "Losses";
5       while( $sth->fetch()){
         # Fetch a row and return column values as scalars
             printf "   %-25s%3d%8d\n",$name, $wins, $losses;
        }

    $sth->finish();
    $dbh->disconnect();

    (Output)
    Selected data for teams.

        Name                 Wins   Losses
    Bath Warships              34       3
    Berkeley Bombers           12      19
    Denver Daredevils          23       5
    Littleton's Tigers         14      18
    Middlefield Monsters       2      32
    Palo Alto Panthers         24      17
    Portland Penguins          28      14
    San Francisco Fogheads     24      12
    Sunnyvale Seniors          12      24
```

解释

1. 准备一个 SQL SELECT 语句，并返回一个语句句柄。
2. DBI execute() 方法把查询发送到数据库，并予以执行。
3. 创建三个标量型变量，它们分别对应于 SELECT 语句中列出的三个字段（列）。
4. bind_column 方法负责指定标量型变量的名称。当 fetch() 方法从数据库检索得到结果集时，这些变量名将分别对应于各个字段。（在老版本的 DBI 中，其第一个参数都指定为 undef；例如 $sth->bind_columns(undef, \$name, \$wins, \$losses); 其中 undef 用于表示 null 字段。
5. fetch 方法负责从结果集中检索一行内容，并把各个值分别赋予 bind_column() 方法中作为参数出现的变量名。球队名将自动赋值到 $name，胜利场数将赋值给 $wins，而败北场数则会赋值给 $losses。每次经过循环时，程序都会把下一行中各列的值赋予上述三个变量，如此反复，直到没有更多数据为止。

? 占位符。 占位符（placeholder）表示为 ?，用于优化查询的处理方式。占位符会为查询提供一套表示值的模板，这些值将在以后某个时刻才赋予相应的字段。它们主要用于 SELECT、INSERT、UPDATE 和 DELETE 语句。

当 DBI 准备一条查询时，数据库必须计划好如何才能最优地处理该请求。语句句柄用于保存已经准备好的查询计划，又称“执行计划（execution plan）”。一般而言，当某个查询执行完毕后，其相应计划就会丢弃。不过如果使用了占位符的话，数据库将不会丢弃执行计划，而是在某个模板中接受该占位符，并围绕它制订计划——这就使得模板能用到以后的查询中。

? 负责表示值，譬如：name=?，其中 name 是数据库表中的字段名，而它的值将在以后提供。（请牢记，? 只能表示值，而不是字段名：?="John" 是错误的！）execute() 方法会接受表示占位符的值的参数，每当调用该方法时，这些值就会在准备好的计划中替换掉相应的占位符。

使用占位符

示例 17.35

```
    use DBI;
    my $driver="DBI:mysql";
    my $database="sample_db";
    my $user="root";
    my $host="localhost";

    my $dbh =
DBI->connect("$driver:$database:$host;user=$user;
  password=quigley1")or die "Can't connect: ". DBI->errstr;
1  my $sth=$dbh->prepare("SELECT name, wins, losses FROM teams
      WHERE name = ?") or
    die "Can't prepare sql statement" . DBI->errstr;

   print "Enter the team name: ";
2  chomp($team_name=<STDIN>);
3  $sth->execute($team_name);
# The value of $team_name replaces the ?
   print "\nSelected data for team \"$name\".\n\n";

4   while(my @val = $sth->fetchrow_array()){
         print "name=$val[0]\n";
         print "wins=$val[1]\n";
         print "losses=$val[2]\n\n";
}
   $sth->finish();
   $dbh->disconnect();

(Output)
Enter the team name: Chico Hardhats

Selected data for team "Chico Hardhats".

name=Chico Hardhats
wins=18
losses=6
```

解释

1. SELECT 语句含有一个带着？占位符的 WHERE 子句，该占位符表示晚些时候将赋值给 name 字段的值。准备该语句，并返回一个语句句柄。
2. 要求用户输入一个球队名称，并赋值给 $team_name。后者将用作 execute 方法的一个参数。
3. 在查询中将 $team_name 的值插入到占位符处。
4. fetchrow_array() 方法负责检索某支球队的值，该球队是在执行第 3 行的查询时确定的。

使用多个占位符

示例 17.36

```
 use DBI;
 my $dbh=DBI->connect("DBI:mysql:host=localhost;user=root;
 password=quigley1;database=sample_db");
```

```
1   my $sth=$dbh->prepare("INSERT INTO teams(name, wins, losses)
                   VALUES(?,?,?)");

    # Preset the values in variables
2   my $team_name="Denver Daredevils";  # set values here
    my $wins=18;
    my $losses=5;
3   $sth->execute($team_name, $wins, $losses);
    print "\nData for team table. \n\n";
4   $sth=$dbh->prepare("SELECT * FROM teams");
    $sth->execute();
5    while(my @val = $sth->fetchrow_array()){
          print "name=$val[0]\n";
          print "wins=$val[1]\n";
          print "losses=$val[2]\n\n";
        }

    $sth->finish();
    $dbh->disconnect();
```

解释

1. 这里的三个占位符起到模板的作用，表示将在晚些时候填入 SQL INSERT 语句中的值。各个 ? 分别表示 name 字段、wins 字段和 losses 字段的值。
2. 将标量赋值为将在执行 SQL 语句时发送给数据库的值。
3. execute() 方法把这些变量值插入到在 INSERT 语句中能够找到的占位符处，并执行 SQL 语句。
4. 准备另一个 SQL 语句，该语句将选取数据库中所有的字段，以便用户检查是否真的插入了新数据。
5. 从上述查询的结果集中每次获取一行内容，并予以显示。

使用占位符插入多条记录

示例 17.37

```
     use DBI;
     my $dbh=DBI->connect("DBI:mysql:host=localhost;user=root;
     password=quigley1;database=sample_db");

     # Using a placeholder. Values will be assigned later

1   my $sth=$dbh->prepare("INSERT INTO teams(name, wins, losses)
                    VALUES(?,?,?)");
    # Create a list of new entries
2   my @rows = (['Tampa Terrors', 4, 5],
                 ['Arcata Angels', 3 , 4],
                 ['Georgetown Giants', 1 ,6],
                 ['Juno Juniors', 2, 7],
                );

3   foreach my $row (@rows ){
      $name = $row->[0];
      $wins = $row->[1];
      $losses=$row->[2];
4     $sth->execute($name, $wins, $losses);
```

```
    }
    print "\nData for team table. \n\n";
5   $sth=$dbh->prepare("SELECT * FROM teams");
    $sth->execute();
         while(my @row = $sth->fetchrow_array()){
           print "name=$row[0]\n";
           print "wins=$row[1]\n";
           print "losses=$row[2]\n\n";
         }

    $sth->finish();
    $dbh->disconnect();
```

解释

1. 三个占位符仍旧起到模板的作用，表示将在晚些时候填入 SQL INSERT 语句中的值。各个 ? 分别表示 name 字段、wins 字段和 losses 字段的值。
2. 创建一个含有行的数组，负责表示那些将在晚些时候插入数据库的新记录。
3. 将来自 @rows 的每一行拆分为各自独立的字段和标量值，这些赋给标量的值分别表示每个相应字段的值。
4. execute() 方法把这些变量值插入到在 INSERT 语句中能够找到的占位符处，并执行 SQL 语句。它会反复处理每一行新数据，直到所有数据都输入完毕为止。如果出现了重复的球队，由于其 name 字段已经赋为主键，因此 execute() 方法会执行失败。

绑定参数和 bind_param() 方法。 另一种方便而高效的使用占位符的途径是 bind_param() 方法。占位符会告诉数据库：由 ? 表示的值将在晚些时候予以填入。限定参数（bound parameter）是用于替换 ? 的值，从而不必再向 execute() 方法发送参数。

bind_param() 方法最多接受三个参数。其中第一个参数表示参数在占位符中的位置；譬如，当位置为 1 时，表示该参数值将填入第一个 ?（占位符）；如果位置为 2，则表示第二个 ?，依此类推。bind_param() 的第二个参数是将要替换 ? 的实际值。最后一个参数是可选的，表示替换值的数据类型，一般都是数字或字符串。在首次调用 bind_param() 之后，其占位符的数据类型就不能再更改了。不过，用户可以不指定该参数，这时它会默认设置为前一个值。

处理数据类型的方法有两种，匿名散列和 DBI 常量：

```
$sth->bind_param(1, $value, { TYPE => SQL_INTEGER }); # Hash
$sth->bind_param(1, $value, SQL_INTEGER);  # DBI Constant
```

有关如何使用第三个参数，请看示例 17.44。

示例 17.38

```
use DBI;
my $driver="DBI:mysql";
my $database="sample_db";
my $user="root";
my $password="quigley1";
my $host="localhost";

my $dbh = DBI->connect("$driver:$database:$host","$user",
"$password") or die "Can't connect: " . DBI->errstr;
```

```
1   $sth=$dbh->prepare("SELECT name, wins,losses FROM teams
    where name LIKE ?") or die "Can't prepare sql statement" .
DBI->errstr;

2   $sth->bind_param(1, "Ch%");
3   $sth->execute();
4   $sth->dump_results();
    $sth->finish();
    $dbh->disconnect();

(Output)
'Cheyenne Chargers', '6', undef
'Chico Hardhats', '21', '25'
```

解释

1. 准备一个含有占位符的 SQL 语句，该占位符将充当查询的模板。
2. bind_param() 方法接受两个参数：占位符的位置（即表示将会填充第几个 ?），以及将会赋值给该位置的值，即“ch%”
3. 由于 bind_param() 已经把参数限定到语句上，因此 execute() 方法无需任何参数。
4. DBI 函数 dump_results() 方法用于在执行完查询后快速输出由数据库返回的结果。

缓存查询。 缓存（cache）是一种临时存储区域，常常用于加快数据的复制或访问速度。大多数数据库都利用缓存来提升近期执行的查询的性能。在执行完一条 SQL 语句后，可以将它缓存起来，而不是销毁掉。如果需要执行与缓存语句一致的另一条查询的话，则可重复使用缓存的查询语句。DBI prepare_cache() 方法用于缓存一条查询。它类似于 prepare() 方法，所不同的是它会查看以前是否已经执行过相同的 SQL 语句。如果是，它就会提供缓存的语句句柄，而不是创建新的句柄。（如果用户需要管理多个连接，请参阅 Apache::DBI::Cache。）

示例 17.39

```
    use DBI;
    my $driver="DBI:mysql";
    my $database="sample_db";
    my $host="localhost";
    my $user="root";
    my $password="quigley1";
    my$dbh=DBI->connect("$driver:database=$database;
    host=$host;user=$user;password=$password")or
         die "Can't connect: " . DBI->errstr;

1 sub get_wins{      # Subroutine to handle database query
2      my($dbh, $team) = @_;
3      my $sth=$dbh->prepare_cached("SELECT wins FROM teams
             WHERE name = ?") or
       die "Can't prepare sql statement" . DBI->errstr;

4      $sth->execute($team);

       $wins=$sth->fetchrow_array();
5      return $wins;
  }
```

```
     STARTOVER: {
6       print "To see how many wins, please enter the team's name. ";
        chomp($team_name=<STDIN>);
             # Call a function to process database query
7       print "$team_name has won ", get_wins($db, $team_name),
             "   games.\n";
        print "Do you want to check wins for another team? ";
        chomp($ans = <STDIN>);
8       redo STARTOVER if $ans =~ /y|yes/i;
   }
   $sth->finish();
   $dbh->disconnect();

(Output)
5 To see how many wins, please enter the team's name. Tampa Terrors
  Tampa Terrors have won 3 games.
7 Do you want to check wins for another team? y
5 To see how many wins, please enter the team's name. San Francisco
Fogheads
 San Francisco Fogheads have won 24 games.
7 Do you want to check wins for another team? y
5  To see how many wins, please enter the team's name. Chico Hardhats
  Chico Hardhats have won 21 games.
7 Do you want to check wins for another team? n
```

解释

1. 使用一个名叫 get_wins 的用户自定义函数来处理数据库请求。
2. @_ 含有两个值，数据库句柄，以及数据库中某支球队的字段名。
3. 准备一个语句。为了提高效率，在执行该语句之后缓存它，而不是销毁它。如需反复执行相同查询的话，该机制就会让处理过程更为高效。鉴于这个函数可能会用到多次，因此这里使用了 prepare_cache() 方法。除了名称不同外，该方法基本类似于 prepare 方法。
4. 执行该查询，并将出现在 SQL 语句中的 ? 填写为球队名称值。
5. 该函数负责检索并返回指定球队的胜利场数。
6. 在程序主体部分中，键入一个标记块，并要求用户选择一支球队。
7. 在 print 语句中调用用户自定义的函数 get_wins，并把数据库句柄和用户选择的球队名称传递给该函数。
8. 如果用户希望看到另一支球队的胜利场数，程序控制流会返回到标记块的顶部，并再次开始处理。

17.5 不返回数据的语句

do() 方法

do() 方法用于在一步之内准备并执行那些无需选择也无需重复的语句。可以使用 do 方法的 SQL 语句包括诸如 UPDATE、INSERT 或 DELETE 之类的语句。这些语句都能修改数据库，同时不返回任何数据。与 prepare 方法不同，do 不会返回一个语句句柄，而是会返回受到影响的行数，或者在查询失败时返回 undef。do() 方法会返回受到影响的行数，或者在查询失败时返回 undef。（返回值为 -1 说明行数不可知、不可获得、或着不可使用。）

```
$rows_affected = $dbh->do("UPDATE your_table SET foo = foo + 1");
```

惟一的缺点是它的性能，尤其是在通过占位符来多次反复执行某个操作时，正如在示例 17.39 中一样。这是因为每个查询都不得不一再重复执行准备、执行等步骤。

添加记录项（entry）。 如需把记录项加入到数据库中的一张表内，可在 DBI 的 do 方法中使用 SQL INSERT 语句。do 方法将返回新记录项的数目，或者在失败时返回 undef。

示例 17.40

```
    use DBI;
    my $dbh=
         DBI->connect("DBI:mysql:host=localhost;user=root,
                    password=quigley1;
                    database=sample_db");
    # Add two new entries
1   $dbh->do("INSERT INTO teams(name,wins,losses)
               VALUES('San Francisco Fogheads', 24,12)");

2   $dbh->do(qq/INSERT INTO teams(name, wins, losses)
          VALUES(?,?,?)/, undef,"Middlefield Monsters", 2, 32);

    $dbh->do(qq/INSERT INTO teams(name, wins, losses)
          VALUES(?,?,?)/, undef,"Littleton's Tigers", 4, 18);

3   $dbh->do("INSERT INTO coaches
               VALUES('','Roger Outback','San Francisco Fogheads',
                      'Defensive Coach','2006-03-16'");

   my $dbh->disconnect();
```

解释

1.2.3. DBI do 方法用于将值插入到 sample_db 数据库的 teams 表中。这里没有出现 prepare 和 execute 方法，因为 do 已经代劳了。它会返回影响到的行数。

删除记录项。下面这个示例会在某些条件满足时删除一条记录。由于 delete 方法不能返回结果集，因此可通过 DBI do 方法来调用它。

示例 17.41

```
    use DBI;
    my $driver="DBI:mysql";
    my $database="sample_db";
    my $user="root";
    my $host="localhost";

    my $dbh = DBI->connect("$driver:database=$database;
        host=$host;user=$user") or die "Can't connect: " . DBI->errstr;

    print "Enter the team name you want to delete: ";
    chomp($name=<STDIN>);
1   my $sth=$dbh->prepare('SELECT count(*) from teams WHERE name = ?');
2   $sth->execute($name);
3   print "Number of rows to be deleted: ", $sth->fetchrow_array(), "\n";
```

```
   print "Continue? ";
   chomp($ans = <STDIN>);
   $ans=lc($ans);
   if ( $ans =~ /y|yes/){
4      $num=$dbh->do(qq/DELETE from teams WHERE name = ?/, undef,
       $name);
5      print ($num > 1 ?"$num rows deleted.\n":"$num row deleted.\n");
   }
   else {
       die "You have not chosen to delete any entries. Good-bye.\n";
   }
    $sth->finish();
    $dbh->disconnect();

(Output)
Enter the team name you want to delete: Sunnyvale Seniors
Number of rows to be deleted: 1
Continue? y
1 row deleted.
```

解释

1. 根据用户输入，将待删除的球队名称赋值给 $team。SQL 语句将通过 count 函数查询数据库，以便获知有多少行匹配于所选的球队名称。
2. execute() 方法将把查询发送到数据库，并返回匹配于所选球队名称的行数。
3. 获取查询结果。这就为用户提供了删除这些记录项的绝好机会。如果不存在任何匹配的球队，则无需继续处理。
4. DBI do() 方法用于准备并执行 SQL DELETE 语句。
5. 返回删除掉的行数。

更新记录项。 如需更新或编辑某个数据库记录项，可通过 DBI do() 方法来使用 SQL UPDATE 语句。

示例 17.42

```
   use DBI;
   my $driver="DBI:mysql";
   my $database="sample_db";
   my $user="root";
   my $password="quigley1";
   my $host="localhost";

   my $dbi=
DBI->connect("$driver:database=$database;host=$host;user=$user;
password=$password")or die "Can't connect: " . DBI->errstr;

   my $num_of_wins;
   my $num_of_losses;
   my $count;
1  print "What is the name of the team to update? ";
   chomp($team_name=<STDIN>);

   # Show user the table before he tries to update it
2  my $sth=$dbi->prepare(qq/SELECT * FROM teams
```

```
      WHERE name="$team_name"/) or die "Select failed: ". $DBI::errstr;
    $sth->execute() or die "Execute failed:".$DBI::errstr;
3   while(($name, $wins, $losses) = $sth->fetchrow_array()){
4     $count++;
      print "\nData for $team_name before update:\n"if $count == 1;
      print "\t\twins=$wins\n";
    print "\t\tlosses=$losses\n\n";
    }
5   if ($count==0){ die "The team you entered doesn't exist.\n";}
6   print "How many games has $team_name won since the last update?";
    chomp($num_of_wins=<STDIN>);
7   print "How many games has $team_name lost since the last update? ";
    chomp($num_of_losses=<STDIN>);

8   $dbi->do(qq/UPDATE teams SET wins=wins+$num_of_wins
      WHERE name = ? /, undef, "$team_name") or
      die "Can't update teams :". DBI->errstr;

9   $dbi->do(qq/UPDATE teams SET losses=losses+$num_of_losses
      WHERE name = ? /, undef, "$team_name") or
      die "Can't update teams :". DBI->errstr;

    # Show the user the table after it is updated
    print "\nData for $team_name after update:\n";
10  $sth=$dbi->prepare(qq/SELECT * FROM teams WHERE
    name="$team_name"/);
    $sth->execute();
    while(($name, $wins, $losses) = $sth->fetchrow_array()){
      print "\t\twins=$wins\n";
      print "\t\tlosses=$losses\n\n";
    }

    $sth->finish();
    $dbi->disconnect();

(Output)
What is the name of the team to update? Chico Hardhats
Data for Chico Hardhats before update:
                wins=15
                losses=3
How many games has Chico Hardhats won since the last update? 1
How many games has Chico Hardhats lost since the last update? 2
Data for Chico Hardhats after update:
                wins=16
                losses=5
```

解释

1. 要求用户输入将在 teams 表中编辑的球队名称。
2. 执行一条 SELECT 语句，检索 teams 表中所有的数据。
3. 在执行更新操作前，首先以当前状态显示该表的内容。
4. 计数器负责跟踪返回的记录总数。
5. 如果计数器值为 0，说明没有结果从 SELECT 处返回，程序会报错并退出。
6. 要求用户输入从上一次更新操作起打赢的球赛场数。
7. 同时要求用户输入从上一次更新操作起败北的球赛场数。

8. 使用 DBI do 方法准备并执行 SQL UPDATE 语句。它会返回更新操作影响到的行数。该语句将更新 teams 表中的 wins 列。
9. 这次更新与上一次类似，所不同的是它会递增败北的场数。
10. 在更新完毕数据库表之后，重新执行这个 SELECT 语句，以便向用户显示编辑之后表的内容。

17.6 事务

在一个有关 teams 表的简单示例中，如果在向两支球队插入数据时，其胜利场数和败北场数突然发生了变化，那么其更新操作将必须修改两支球队信息，而不是一支。假定用户正在更新多个表，那么其更新操作可能会在某张表中执行成功，同时在其他表中执行失败。一个典型的例子是从某张表中的储蓄账户里提取现金，并存入另一张表中的核算账户。如果存款操作成功，但取款失败的话，这些表就会陷入不一致的状态。事务（transaction）是一组 SQL 语句，它们作为同一个执行单元，要么执行成功，要么全部失败。例如，INSERT、UPDATE 和 DELETE 语句可以放在同一组中执行。如果其中一个语句失败的话，则其他所有语句都不会执行。

在默认情况下，MySQL 会把 autocommit 模式设置为打开（on）状态。DBI 在默认情况下也开启了 autocommit 模式。这就是说，一旦用户执行完任意一条修改表的语句，同时没有返回任何错误的话，MySQL 就会立刻将该语句提交给数据库，并持久化任何影响到数据表的操作。

如需使用 MySQL 中的事务机制，则必须关闭其 autocommit 模式。在 Perl 脚本中，用户可以通过在连接数据库时设置散列值 AutoCommit=>0 的方式达成该效果，正如示例 17.43 所示。

在前面展示的示例中，当用户连接到某个数据库时，connect() 方法用到了有关错误处理的散列选项，即 PrintError 和 RaiseError。而在使用事务时，用户必须关闭 AutoCommit 属性，打开 RaiseError 属性，并把可选的 PrintError 属性设置为“on”或“off”，其中“on”是默认情况。

示例 17.43

```
1 my $dbh = DBI->connect( 'dbi:mysql:sample_db','root','quigley1',
                          {
                          PrintError => 0,
                          RaiseError => 1,
2                         AutoCommit => 0
                          }
```

RaiseError 告诉 DBI，如果碰到错误的话，就打印 $DBI::errstr 消息并退出程序；而 PrintError（默认是打开的）将促使 DBI 借助 $DBI::errstr 发送一条警告信息，并让程序继续执行。

提交和回滚。所谓提交（commit），指的是在事务中执行一组语句，并将它们作为同一个组（group）发送给数据库。如果其中所有语句都执行成功，则提交这个组，从而完成数据库的修改。但是，如果组内任意一个语句发生错误的话，就会引发一次回滚（rollback）命令，并将所有的表退出到以前的状态。

当在 Perl 中处理事务时，往往会利用 eval 块来跟踪错误，然后使用 commit() 或 rollback() 方法来完成事务。

下面这个示例将把一组记录插入到表中。如果在添加这些记录项的过程中发生错误，则整个过程都将会回滚。这里发生错误是很常见的事，因为可能某个记录项已经存在了。

示例 17.44

```
1  use DBI qw(:sql_types);

2  my $dbh = DBI->connect('dbi:mysql:sample_db','root','quigley1',
                          {
                            PrintError => 0,
3                           RaiseError => 1,
4                           AutoCommit => 0
                          }
         ) or die "Connection to sample_db failed: $DBI::errstr";
5  my @rows = ( # New rows to be inserted
                 [ 'Tampa Terrors', 3, 5 ],
                 [ 'Los Alamos Lizzards', 12, 3 ],
                 [ 'Detroit Demons', 22, 0 ],
                 [ 'Cheyenne Chargers',6, 0 ],
          );
6  my $sql = qq{ INSERT INTO teams VALUES ( ?, ?, ? ) };
7  my $sth = $dbh->prepare( $sql );
8  foreach $param (@rows) {
9    eval { # The eval block is used to catch errors
10   $sth->bind_param( 1, $param->[0], SQL_VARCHAR );
     $sth->bind_param( 2, $param->[1], SQL_INTEGER );
     $sth->bind_param( 3, $param->[2], SQL_INTEGER);
     $sth->execute() or die;
     };
   }
11 if( $@ ) { # If eval failed. $@ is set to the error that occurred
     warn "Database error: $DBI::errstr\n";
12   $dbh->rollback(); # Reverse all commit statements
   }
   else{
13   $dbh->commit();
     print "Success!\n";
   }
   $sth->finish();
   $dbh->disconnect();
```

解释

1. 借助特殊的 DBI::sql_types 标签，引入一些表示 SQL 标准类型值的常量。第 10 行处的 bind_param 方法将用到这些常量。
2. 产生一个指向 MySQL 数据库的连接。
3. 打开 RaiseError 属性，以便捕获异常，并在捕获后退出程序。
4. 关闭 AutoCommit 属性，从而让 SQL 语句不会自动发送到数据库，而是必须手动提交。
5. 创建一个匿名数组列表，用于表示将要插入表中的各行内容。
6. 创建一个 SQL 语句，以便在晚些时候插入新的球队；其值将替换其中的 ? 占位符。
7. 准备 SQL 语句。返回一个语句绑定。
8. 使用 foreach 循环遍历将要添加的每一行内容。
9. 进入 eval 块。如果碰到错误的话，就把它赋值给特殊变量 $@，见第 11 行。
10. bind_param() 方法将第一个参数绑定到了第 6 行中的首个占位符（即 ?）上。当第一次进入循环时，其第一个参数 $param->[0] 是‘Tompa Terrors’。其类型是 SQL_VARCHAR。此后，第二个参数 param->[1] 则会绑定到第二个占位符（即 ?）上。当第一次通过循环时，它表示胜利的次数；譬如三次。依此类推。
11. 如果 eval 块中某一条语句执行失败的话，该变量将设置为错误的内容。
12. 如果 eval 块中的任意一条语句执行出错的话，所有这些语句都将丢弃。数据库将回滚为其

原始状态。换而言之，如果某个事务失败的话，什么都不会发生。

13. 如果没有出错，则执行该语句块，其中所有的插入动作都将会发生。

17.7　使用 CGI 和 DBI 选择并显示记录项

在学会了如何连接数据库并提交查询之后，用户可能就想在自己的 CGI 脚本中使用 DBI 来为动态 Web 页面存储并检索数据。下面第一个示例通过一段简单的 CGI 程序展示了如何从数据库中获取数据，然后把它们格式化为一张表，并发送到用户的浏览器上。

示例 17.45

```
1 #!c:/ActivePerl/bin/perl.exe
2 use DBI;
3 use CGI qw(:standard);


4 my $dbh =
DBI->connect("DBI:mysql:host=localhost;database=sample_db;
    user=root;password=quigley1")or die "Connection to sample_db
failed:  $DBI::errstr";
;

5 my $sth=$dbh->prepare("SELECT * FROM coaches");
  $sth->execute();

6  print header, start_html(-title=>"Sample Database",
                      -BGCOLOR=>"#66ff33");
   print "<div align='center'>";
   print h2("Contents of the \"coaches\" Table");

7  print qq/<table border="1" cellpadding="10" bgcolor="white">/;
8  while(my @val = $sth->fetchrow_array()){
        print <<EOF;
         <tr>
              <td>$val[0]</td>
              <td>$val[1]</td>
              <td>$val[2]</td>
              <td>$val[3]</td>
              <td>$val[4]</td>
         </tr>
EOF
  }
   print "</table>";
   print end_html();
   $sth->finish();
   $dbh->disconnect();
```

解释

1. shbang（镑符号，bang 行）把 Perl 的安装位置提供给 Web 服务器，以便后者启动正确的 Perl 解释器。不管在什么操作系统上使用 Apache 服务器，本行都是必需的。
2. 在这个程序中，我们将使用 DBI 模块访问 MySQL 数据库。
3. 我们将使用面向函数版本的 CGI 模块，以便使用标准 CGI 函数来访问 Web 浏览器。
4. 生成一个指向 MySQL 数据库的连接。
5. 准备一条 SQL SELECT 语句。它将为 coaches 表检索所有的记录。

6. CGI header 函数设置了“Content-type: text/html\n\n”行；而 HTML 首部则负责设置其标题和 HTML 页面颜色。这些信息将通过通用网关（Common Gateway）发送给服务器，进而发送到浏览器上。
7. 创建一个 HTML 表，它持有数据库返回的数据。
8. 当获取到来自数据库的数据时，首先对它们进行格式化，并发送回服务器，然后在浏览器窗口中把它们显示为一张表。

下面第二个示例则为我们提供了一整套流程。

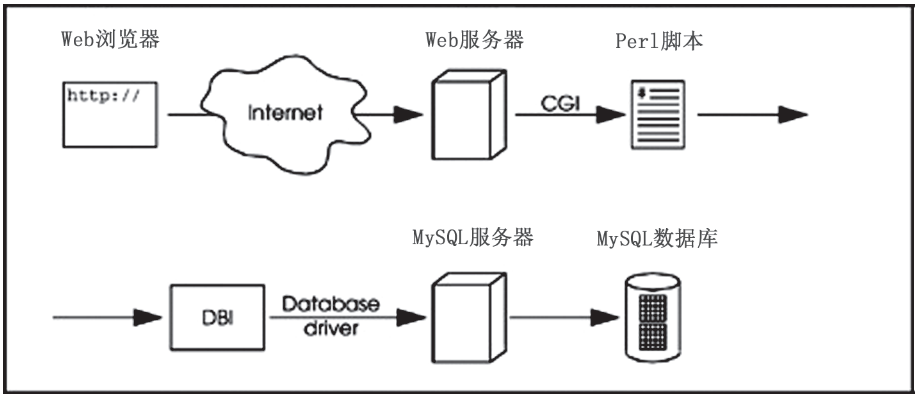


图 17-21　从浏览器到数据库的客户端 / 服务器模型

CGI.pm 创建了一个由文本字段构成的简单 HTML 表单。用户将选择一支球队，把球队名称键入表单，然后提交这个表单。CGI 程序将处理表单信息，使用 DBI 模块连接到 MySQL 数据库，检索球队信息，进行格式化，并以一个 HTML 表格的形式发送回浏览器。

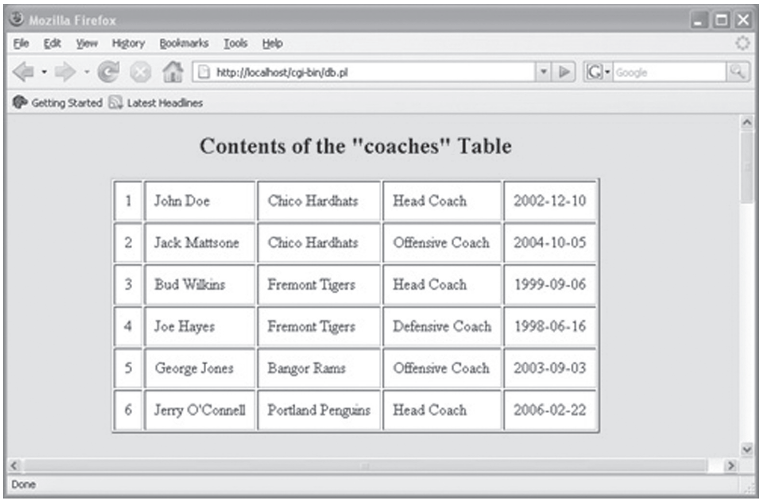


图 17-22　DBI 与 CGI 输出

示例 17.46

```
1    #!c:/ActivePerl/bin/perl.exe

2    use DBI;
3    use CGI qw(:standard);

     print header, start_html(-title=>"Team Lookup",
                              -BGCOLOR=>"#66ff33"
                             );
4    print start_form,"<font face='arial' size='+1'>
     Look up what team? ",textfield('name'),p;
5    print submit, end_form, hr;

6    if(param())  {
7        $team = param('name');

8        $dbh =
         DBI->connect("DBI:mysql:host=localhost;database=sample_db;
         user=root;password=quigley1") or
         print "Connection failed: ". $DBI::errstr;
9        $sth=$dbh->prepare("SELECT name, wins, losses
                             FROM teams where name = ?"
                           );
         $sth->execute($team);
10       if ($sth->rows == 0){
             print "Your team isn't in the table.<br>";
             exit;
         }
         print h2("Data for \u$team");
         while(($name,$wins,$losses) = $sth->fetchrow_array()){
11           print <<EOF;
             <table border="1" bgcolor="yellow">
                <tr>
                    <th>Name</th>
                    <th>Wins</th>
                    <th>Losses</th>
                </tr>
                <tr>
                    <td>$name</td>
                    <td>$wins</td>
                    <td>$losses</td>
                </tr>
             </table>
12           EOF
             print end_html();
             $sth->finish();
             $dbh->disconnect();
         } # End while loop
     }  # End if block starting on line 6
```

解释

1. shbang（镑符号，bang 行）把 Perl 的安装位置提供给 Web 服务器，以便后者启动正确的 Perl 解释器。不管在什么操作系统上使用 Apache 服务器，本行都是必需的。
2. 在这个程序中，我们将使用 DBI 模块访问 MySQL 数据库。
3. 我们将使用面向函数版本的 CGI 模块，以便利用标准 CGI 函数来访问 Web 浏览器。

4. 该函数开始一个 HTML 表单。该表单非常简单。它仅由一个 HTML 文本字段构成，用户将在其中输入数据。
5. CGI submit 函数负责创建表单的 Submit 按钮。当用户按下 Submit 按钮时，表单就会提交，同时将输入到文本字段中的数据发送给 CGI Perl 脚本，以便进行进一步地处理。
6. 由于 CGI 模块将这段 Perl 脚本赋予了 HTML 表单中的 ACTION 属性，因此该 Perl 脚本将负责接收并处理表单数据。CGI 模块将会检查其是否有参数，也就是说检查表单是否返回了数据？如果 params 函数返回真（true），就说明表单已经提交，这段程序将继续执行。如果 param 函数返回假（false）的话，则说明该表单尚未提交。
7. HTML 表单中的文本字段名为“name”。CGI param 函数以文本字段名为参数，并能返回赋给它的值；在这里就是球队的名称。
8. 连接到 MySQL 数据库，使用 sample_db 数据库。
9. 准备一条带有占位符的 SQL 语句，该占位符将在晚些时候赋值为 teams 表中某支球队的名称。
10. 执行查询，并发送用户在填写表单时选中的球队名称。
11. 当数据库返回结果集之后，启动一个 here doc。在这个 here doc 中创建一个 HTML 表格，以便在浏览器中以良好的结构格式显示来自数据库的结果集。CGI 将把该表格发送给 Web 服务器（Apache），进而显示到浏览器（Firefox）上。
12. 这个用户定义的终止符表明 here doc 到此结束。请记住，该标记必须位于 Perl 脚本的最左侧一列，并且其后不能出现空格；事实上，它的后面必须以一个换行符结尾。

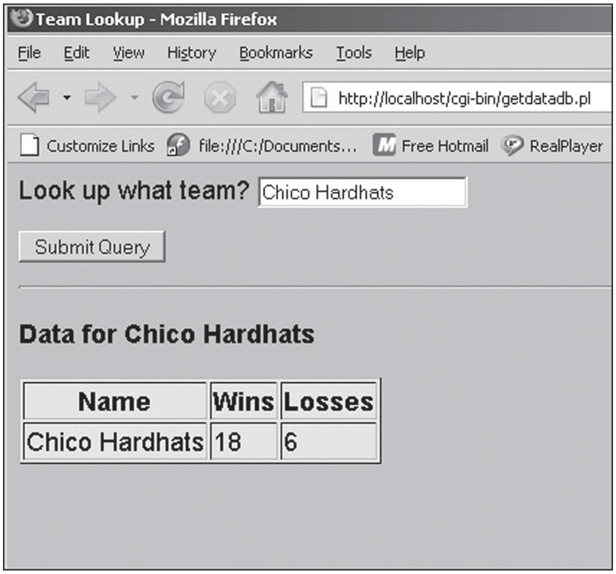


图 17-23 CGI、DBI 与一个表单

17.8 还有哪些内容

本章介绍了 MySQL 关系型数据库，以及如何在 mysql 客户端中执行 SQL 语句。在学习了命令行下的基本查询之后，本章接着介绍了 Perl DBI 模块，以便用户能在 Perl 脚本中执行相同的 mysql

函数。我们讨论了最为常用的 DBI 方法，以及如何使用这些方法连接数据库、选择和检索数据、更新并移除记录，等等。最后我们更进一步，创建了一段 CGI/Perl 程序，并使用 DBI 模块生成了一个动态 Web 页面，该页面能与来自表单的用户请求进行交互，并通过从数据库中检索到的数据予以响应。

尽管我们已经学习了 DBI 的主要内容，但前面还有很多知识等待去发掘。如需搜索有关 DBI（包括其方法、变量、常量、函数等）的详细文档，读者可在如图 17-24 所示的 CPAN 知识库中找到最佳的资源。

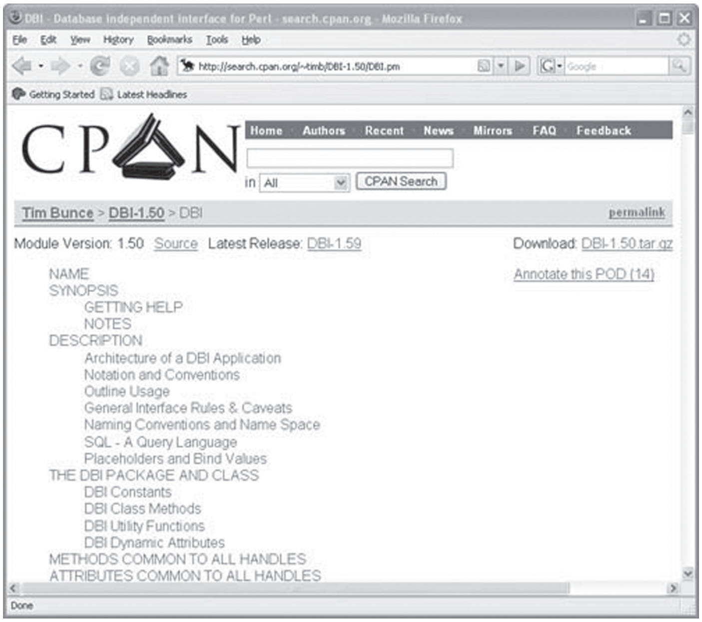


图 17-24 CPAN 知识库

17.9 读者应当学到的知识

1. 什么是关系型数据库？
2. 什么是 MySQL ？
3. 从哪里可以获得 MySQL ？
4. 什么是 MySQL 客户端？
5. 如何为 MySQL 设置口令？
6. test 数据库有什么用？
7. mysql 数据库有什么用？
8. 什么是 SQL ？
9. 什么是 DBI ？如何获取它？
10. 什么是 DBI-MySQL ？
11. 什么是数据库句柄（database handle）？
12. 什么是语句句柄（statement handle）？

13. Perl 是如何查询数据库的？结果集在哪里结束？

14. 在为 Perl 的 prepare 方法准备查询（query）时，它与在 MySQL 客户端中生成的查询是否一样？

15. 如何从 Perl 程序中检索结果集？

16. 什么是缓存查询（cached query）？

17. 占位符（placeholder）有什么用？

18. 如何获知数据库连接是否成功？

19. 如何从 Perl 脚本中关闭数据库？

20. Tim Bunce 是谁？

21. 哪里可以找到有关 Perl DBI 的入门教程？

22. CGI 和 DBI 是如何协作的？

17.10 下章简介

下一章将讨论 Perl 脚本如何通过执行系统调用来实现与操作系统的交互，并将介绍一些 Perl 函数，负责处理目录、权限、属主、硬链接与软链接、文件重命名、获取文件统计信息等功能。读者将学习如何在基于 UNIX 和基于 Windows 的系统上使用这些函数。

练习 17 Select * from Chapter

第一部分 SQL 实验室——使用“northwind”数据库

1. 从一段脚本中载入“northwind”数据库。
2. “northwind”数据库对于 Microsoft 的 Access Database 而言是一个优秀的示例数据库，它经过裁剪后也适用于 MySQL。
3. 本练习为读者提供了一个“northwind.sql”脚本（在华章网站 www.hzbook.com 上），该文件含有创建并填充 northwind 数据库的 SQL 语句。该文件必须位于正在运行 mysql 的客户端主机上。
4. 为执行该脚本，请读者登录 MySQL 控制台窗口，并键入如下内容：

   source c:\document\northwind.sql

   （请注意，文件名两边是没有引号的，绝对和相对路径名在这里都可使用。）

   执行 SQL 脚本的其他方法：

   ```
   mysql> SOURCE C:\path\northwind.sql;
   mysql> SOURCE ..\path\northwind.sql;
   shell> mysql db_name < input_file
   ```

   当脚本执行时，其结果将显示在屏幕上，它们大部分都是 SQL“insert”语句。当碰到错误时，便会退出处理流程，并且不返回任何值。

   详见： http://showroom.declarativa.com/northwind.htm

练习 SQL 命令

1. 在执行完 northwind.sql 脚本后，首先键入“use northwind”。
2. 键入如下 SQL 命令，并解释它们都做了什么。

a. SHOW DATABASES;

b. SHOW TABLES FROM NORTHWIND;

c. SHOW FIELDS FROM SHIPPERS;

d. DESCRIBE SHIPPERS。

3. 请读者使用 SQL 入门练习，并执行：

从 Shippers 表中选择所有的行，并显示其字段。

从 Employees 表中选择所有的行，并只显示其 FirstName 与 LastName 字段。

当 Country 值为 Italy 时，从 Customers 表中选择 CompanyName 与 Phone 字段。

按照排列后的顺序从 Customers 表中打印 ContactName 和 Country 字段，并按 Country 排序。

当 Country 值以“Po”或“Sw”开头时，从 Customers 表中选择 ContactName、CompanyName 和 Country 字段。

获取 Products 表中所有产品的数目。

从 Products 表中打印前 10 个产品。

不重复地从 Customers 表中以字母表顺序选择国家。

找出所有介于 $10 和 $20 之间的产品。

插入一个新产品。

更新 Products 表，把 Sir Rodney 的 Marmalade 单价加上 $5.25。

选择所有库存量小于 10 个单位的产品。

删除数据库中最久远的订单。

使用一个 SQL 函数打印当前日期与时间。

第二部分　Perl/MySQL 实验室

咨询 MySQL 文档（mysql.com），从中获取本练习需要的正确数据类型与函数。该文档写得非常出色。

1. 创建一段名叫“school.sql”的 SQL 脚本，实现如下功能：

a. DROP SCHOOL IF EXISTS SCHOOL;

b. CREATE SCHOOL;

c. 创建一个名叫“student”的表，含有如下字段：

FirstName

LastName

Email

CellPhone

Major

GPA

StartDate

StudentID (primary key)

2. 使用上述 school 数据库

a. 向 student 表中插入三行数据。

3. 在 MySQL 提示符下，执行“school.sql”脚本。

使用 SQL 的 describe 语句查看 school 数据库的结构。

4. 选择 student 表的所有行。如果该表含有和预期值一样的列与数据，读者便可准备进入本练习的下一部分。

5. 在一段 Perl 脚本中，使用 DBI 模块实现：
   a. 连接 MySQL，并打开 school 数据库。
   b. 准备一个缓存查询，负责选取 student 表中所有的行，并按形式排序，最后以题头形式显示所有的列。

6.
   a. 创建另一个 Perl 脚本，让用户从菜单中选择如下选项：
      1) Update a record
      2) Delete a record
      3) Insert a new record
      4) Display the table
      5) Exit

      针对菜单中的每个选项，使用 MySQL 函数创建一个负责执行指定任务的子例程。读者可否创建一个 CGI 程序来处理所有这些逻辑呢？
   b. 创建一个循环，不断重新显示上述菜单，直到用户选择 5 为止。
   c. 重写上述程序，关闭 AutoCommit 并打开 RaiseError 属性。在用户选择退出时，才把所有修改操作提交给数据库。

第三部分　回到 Checking.pm

1. 现在，读者不妨重新设计原有的 Checking.pm 模块，让它使用 MySQL 数据库，而不是原本用于跟踪余额信息的文本文件。

   在 mysql 提示符下创建数据库和注册表。该注册表应当含有一些字段，它们匹配于在练习一中创建的“register”文本文件。

   在 Perl 模块 Checking.pm 中打开数据库连接。如需获取余额值，应当从“register”表中选取。当读者调用 exit() 函数时，就会使用 SQL INSERT 命令把前一个含有新信息的事务插入到数据库中。

   创建一个 Perl 函数，它会在调用时显示注册表的内容。

   读者应当了解如何去：

   使用 PPM 和 CPAN 安装 MySQL、DBI 以及数据库驱动。

   执行基本的 MySQL 命令。

   从一段 Perl 脚本中连接到数据库服务器。

   使用 Perl MySQL 函数实现：

   选择一个数据库

   准备并执行 SQL 语句

   使用 DBI 方法遍历结果集

   使用占位符

   绑定列和参数

   检查由数据库服务器产生的错误

   跟踪检索到的记录数目

   丢弃已经准备好的查询

   从一段 Perl 脚本中断开数据库连接

第 18 章　与系统交互

18.1　系统调用

从 shell（批处理）编程转到 Perl 编程的程序员往往希望 Perl 与 shell 一样只是由一组 UNIX（或 MS-DOS/NT）命令组成的序列。然而，Perl 并不像 shell 那样直接访问系统实用程序。当然，为了实现其功能，Perl 必须通过某种途径与操作系统进行交互（interface）。事实上，Perl 提供了一组专门负责与操作系统交互的函数。这些函数与 UNIX 系统调用直接相关，因此读者经常可以在 C 中找到它们。Windows 也支持很多 UNIX 系统调用。本书后面还将列出一些在通常情况下不支持的系统调用。

系统调用（system call）负责向操作系统（内核）请求某些服务，譬如获得当前时间，删除文件，创建新进程，终止进程，等等。最重要的一组系统调用涉及进程的创建与终止，内存的分配和释放，以及向进程发送信息（譬如信号）。其他的系统调用还与文件系统相关，包括：创建文件、读写文件、创建和删除目录、创建链接等㊀。

UNIX ㊁系统调用在 UNIX 手册页的第二部分中已有说明。Perl 的系统函数不论在语法上还是在实现上都几乎与它们相同。如果系统调用失败，它会返回 −1，并将系统全局变量 errno 设置为表明错误原因的值。C 程序可使用 perror 函数获取保存在 errno 中的系统错误信息；而 Perl 程序则使用特殊变量 $!（详见“错误处理”一节）。

当需要获取或处理有关文件或进程的信息时，可使用下面介绍的 Perl 函数执行各种系统调用。如果 Perl 没有提供所需系统调用，则可使用 Perl 的 syscall 函数，该函数以 UNIX 系统调用作为参数（详见“syscall 函数和 h2ph 脚本”一节）。

除了内建的函数外，标准 Perl 库还捆绑有 200 个各类模块，可用于对文件、目录、进程、网络等执行可移植的操作。如果已经安装了 ActiveState 的话，在标准 Perl 库 c:\Perl\site\lib\win32 中还提供了一组 Win32 模块库集合。

读者如需从标准 Perl 库中读取任何模块（其文件名中带有 .pm 扩展名）的相关文档，可使用 Perl 内建的 perldoc 函数或 UNIX 的 man 命令。此外，ActiveState（Win32）提供了在线文档，读

㊀ 系统调用能够直接进入内核，而库调用则是调用系统调用的函数。Perl 的系统接口函数是根据其对应的 UNIX 系统调用的名字而命名的。

㊁ 从现在开始，本书提到 UNIX 的地方也同样适用于 Linux。

者可以按下“开始”->“程序”，然后选择“ActiveState”来找到这些文档。

示例 18.1

```
(At the command line)
$ perldoc Copy.pm
```

解释

perldoc 函数以模块名（带有或不带 .pm 扩展名）作为参数。它能在窗口（在 Win32 平台上是 Notepad 记事本窗口）中显示模块的文档。本示例显示了标准 Perl 库中 Copy.pm 模块的部分文档。

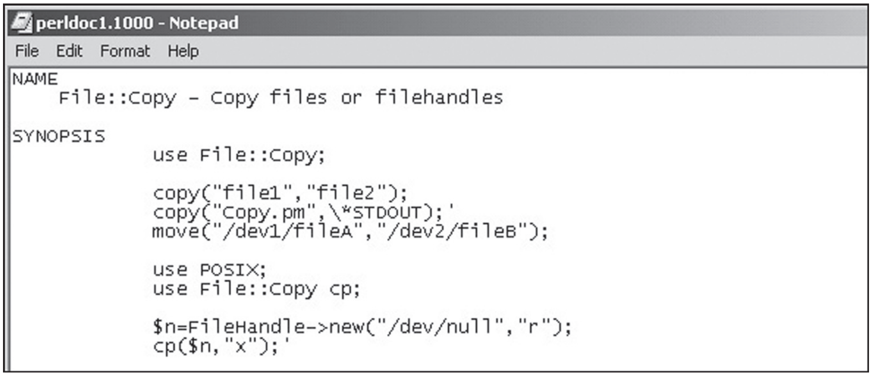


图 18-1　perldoc 和 Copy.pm 模块

18.1.1　目录和文件

在浏览文件系统时，目录都是以斜杠隔开的。UNIX 文件系统使用正斜杠（/）指定根目录，其后是由正斜杠隔开的子目录。如果指定了文件名，它一定是路径的最后一部分。文件名和目录名都是大小写敏感的，由字母和数字字符构成，并且不允许出现空白符号。文件名中的点号（.）没有特殊意义，但可用于隔开基本文件名和扩展名，譬如 program.c 或 file.bak。文件名的长度随着操作系统的不同而不同，最少可以是一个字符，而大多数 UNIX 文件系统的最大允许长度是 255 个字符。只有根目录才能命名为 /（正斜杠）㊀。

Win32 中的文件系统主要包括 FAT、FAT32 和 NTFS，它们使用不同的规范规定目录路径，其中最基本的 FAT 目录和文件都是通过反斜杠（\）而分隔的。其名称是大小写敏感的，最初限制为 8 个字符，其后是一个点号，后缀是不超过 3 个字符的扩展名。（Windows 2000/NT 允许更长的文件名）。其文件系统的根目录是 C:\ 或者 D:\，而不是只有反斜杠。在网络环境中，其通用命名规范（UNC）使用不同的规范来分隔路径中的各部分，譬如在 //myserver\dir\dir 中就将驱动器盘符替换成了两个反斜杠。

反斜杠问题。在 Perl 脚本中，反斜杠一般用作转义符或引用字符（如 \n、\t、\U 和 \$500 等）。因此，在规定 Win32 路径的分隔符时，经常需要使用双反斜杠，除非个别模块允许把单个反斜杠或路径名放到单引号之中。例如，C:\Per\lib\File 应当写作 C:\\Perl\\lib\\File。

File::Spec 模块。 标准 Perl 库中 File::Spec 模块的设计目标是支持那些经常对文件名执行的可

㊀ Mac OS 文件系统（HFS）也是层次结构的，并使用冒号来隔开路径中的各个部分。

移植操作。譬如创建路径组件列表之外的单个路径，并针对相应操作系统应用正确的路径限定符；或者将路径拆分为卷（volumn）、目录和文件名等。表 18-1 列出了所有 File::Spec 函数的列表。

由于这些函数在不同的操作系统上都不相同，因此基于各种 OS 的例程集合都分别归入到不同的模块中。这些模块包括：

File::Spec::UNIX
File::Spec::Mac
File::Spec::OS2
File::Spec::Win32
File::Spec::VMS

表 18-1　File::Spec 函数

函　数	用　途
abs2rel	根据目标路径和可选的起始路径，返回从起始路径到目标路径的相对路径
canonpath	不对文件系统进行物理检查，但是逻辑清除给定的路径。在 UNIX 中还会清除连续的斜杠及其后面的 "/"
case_tolerant	返回真或假值，表明在对比文件规范时是否在意字母的大小写
catdir	连接两个或多个目录名，形成完整的路径，该路径以目录结尾，并从结果字符串中删去位于末尾的斜杠
catfile	将一个或多个目录名与文件名相连，形成以文件名结尾的完整路径
catpath	根据卷、目录和文件部分，返回完整的路径。在 UNIX 系统中会忽略 $volumn，只连接目录和文件。如果需要的话就插入 "/"
curdir	返回表示当前目录的字符串，在 UNIX 上就是“.”
devnull	返回表示空设备的字符串，在 UNIX 上是“/dev/null”
file_name_is_absolute	以路径为参数，如果它是一个绝对路径，则返回真
join	join 等同于 catfile
no_upwards	给出文件名列表，并去除指向父目录的引用
path	不接受任何参数，以数组形式返回环境变量 PATH
rel2abs	将相对路径转换为绝对路径
rootdir	返回表示根目录的字符串，在 UNIX 上是“/”
splitpath	将路径拆分为卷、目录和文件名等各个部分。在没有卷概念的文件系统上，对于卷则返回 undef
tmpdir	返回一个表示随后列表中第一个可写目录的字符串，如果没有可写的目录，则返回“”
updir	返回表示父目录的字符串。在 UNIX 上是“..”

示例 18.2

```
1   use File::Spec;
2   $pathname=File::Spec->catfile("C:","Perl","lib","CGI");
3   print "$pathname\n";

(Output)
3   C:\Perl\lib\CGI
```

解释

1. 如果没有指定操作系统的话，则针对当前操作系统加载 File::Spec 模块，在本例中是 Windows

2000 系统。这是一个面向对象的模块，但它也拥有面向函数的语法。
2. 标量 $pathname 含有的路径是由传递给 catfile 方法的参数所构成的。catfile 函数能够连接一组路径元素。
3. 打印新的路径，并使用反斜杠分隔路径的各个部分。在 UNIX 系统上，路径应当打印为 /Perl/lib/CGI。

18.1.2 目录和文件属性

UNIX。大多数常见的文件类型都属于常规文件（regular file），它们含有数据和有序的字节序列。其数据可以是文本数据或二进制数据。有关文件本身的信息则存储在称作信息节点（inode）的系统数据结构中。信息节点内的信息由属性构成，这些属性包括链接计数值、文件所有者、组、模式、文件大小、最后一次访问时间、最后一次修改时间和文件类型等。用户可使用 UNIX 的 ls 命令查看目录中各文件的信息节点情况。还可使用 stat 系统调用检索这些信息。此外，Perl 的 stat 函数也能给出有关文件的信息。该函数能检索设备号、信息节点号、模式、链接计数值、用户 ID、组 ID、文件字节大小、上一次访问时间，等等（详见“stat 和 lstat 函数”一节）。

目录是由 UNIX 内核维护的特殊文件，其内容是文件名的列表。其中每个文件名都具有一个指向相应文件信息的编号。该编号又称作信息节点编号（inode number），指向相应的信息节点。信息节点中含有文件的相关信息，并指向文件数据块在磁盘上的位置。下面这几个函数负责操纵目录、更改文件权限和创建文件链接。

Windows。文件和目录中都含有描述自身属性的元信息。Win32 文件和目录的四个基本属性分别是 ARCHIVE、HIDDEN、READONLY 和 SYSTEM。详见表 18-2。

Directory Entry	
Inode #	Filename

表 18-2 文件与目录的基本属性

属 性	描 述
ARCHIVE	设置何时更改文件内容
HIDDEN	不显示在目录列表中的文件
READONLY	不能更改的文件
SYSTEM	特殊的系统文件，如 IO.SYS 和 MS-DOS.SYS，它们一般也是不可见的

如要检索并设置文件属性，可以使用标准 Perl 提供的 Win32::File 扩展。除非特别予以说明，否则所有函数在调用失败时都会返回 FALSE(0)。用户应当根据需要把函数名导入到调用程序的命名空间中。详见表 18-3。

表 18-3 Win32::File 函数

函 数	用 途
GetAttributes(Filename,ReturnedAttributes)	获取文件或目录属性。将 ReturnedAttributes 设置为文件名属性的 or 组合
SetAttributes(Filename,NewAttributes)	设置文件或目录属性。必须将 NewAttributes 设置为属性的 or 组合

如需检索文件属性，用户可使用 Win32::File::GetAttributes($Path,$Attributes)；如需设置文件属性，则可使用 Win32::File::SetAttributes($Path,$Attributes)。参见表 18-4。Win32::File 还提供了很多常量，详见示例 18-3。

表 18-4 Win32::File 属性

属性	描述
ARCHIVE	设置文件内容的更改时间。通过备份程序使用
COMPRESSED	Windows 压缩文件，而不是 zip 文件。用户无权设置它
DIRECTORY	该文件是目录。用户无权设置它
HIDDEN	文件在目录清单中不显示
NORMAL	常规文件。不设置 ARCHIVE、HIDDEN、READONLY 和 SYSTEM
OFFLINE	数据不适用
READONLY	不能更改的文件
SYSTEM	特殊系统文件，如 IO.SYS 和 MS-DOS.SYS，一般也是不可见的
TEMPORARY	由某些程序创建的临时文件

示例 18.3

```
use Win32::File;
$File='C:\Drivers';
Win32::File::GetAttributes($File, $attr) or die;
print "The attribute value returned is: $attr.\n";
if ( $attr ){
   if ($attr & READONLY){
      print "File is readonly.\n";
   }
   if ($attr & ARCHIVE){
      print "File is archive.\n";
   }
   if ($attr & HIDDEN){
      print "File is hidden.\n";
   }
   if ($attr & SYSTEM){
      print "File is a system file.\n";
   }
   if ($attr & COMPRESSED){
      print "File is compressed.\n";
   }
   if ($attr & DIRECTORY){
      print "File is a directory.\n";
   }
   if ($attrib & NORMAL){
      print "File is normal.\n";
   }
   if ($attrib & OFFLINE){
      print "File is normal.\n";
   }
   if ($attrib & TEMPORARY){
```

```
            print "File is temporary.\n";
         }
      }
      else{
7        print Win32::FormatMessage(Win32::GetLastError),"\n";
      }

(Output)
4     The attribute value returned is 18.
      File is hidden.
      File is a directory.
```

解释

1. 加载 Win32::File 模块。
2. 将 C:\ 驱动器上的文件夹 Drivers 赋值给 $File。
3. 用两个参数调用 GetAttributes 函数：第一个参数是文件的名字，第二个则是 READONLY、HIDDEN 等属性常量逐位 or 得到的值。该值由函数 GetAttributes 提供。请注意，这里必须以全限定的包名来调用函数 GetAttributes。这是因为该函数位于 Win32::File 模块的 @EXPORT_OK 中，因此必须由用户专门地请求，或者给出其全限定的名字。如果是专门请求的话，则其中所有的常量也应当一并列出，否则就无法将它们切换到用户的命名空间中。
4. 打印对属性值执行或（or）操作后的值。如果该值为 0，说明可能有些东西出错了，则从第 7 行格式化并打印错误信息。
5. 如果文件或目录的属性中有一个存在的话，则通过其后面的检测代码显示它究竟返回了哪一种描述文件或目录的属性。
6. 把 $attr 的值与常量值（在这里是 READONLY）逐位执行逻辑与（and）运算，如果得到的值是真（非 0），则说明该文件是只读的。
7. 该函数将针对 Windows 报告的最后一个错误生成一份可读的错误消息。

18.1.3　查找目录和文件

File::Find 模块能让读者遍历整个文件系统树，并根据某些条件查找指定的文件或目录。它等效于 UNIX 的 find 命令或 Perl 的 find2Perl 翻译器。

格式

```
use File::Find;
find(\&wanted, '/dir1', '/dir2');
sub wanted { ... }
```

find() 的第一个参数是一个指向散列的引用，负责描述将要对文件执行的操作，或是一个指向子例程的引用。用户可输入 perldoc File::Find 查看其详细信息。wanted() 函数负责确认需要查找的文件。$File::Find::dir 含有当前目录名，当位于该目录中时，会自动将 $_ 赋值为当前的文件名。$File::Find::name 含有指向文件的完整路径名。在调用函数时，可使用 chdir() 进入 $File::Find::dir，除非设定了 no_chdir。find() 函数的第一个参数是一个指向散列的引用，负责描述将要对文件执行的操作，或是一个指向子例程的引用。File::Find 的散列引用键如表 18-5 所示。

表 18-5　File::Find 的散列引用键

键	值
bydepth	在报告完毕所有项目后报告目录名
follow	跟随符号链接
follow_fast	类似于 follow，所不同的是它可以同时报告多个文件
follow_skip	只处理一次文件（不包括目录和符号链接）
no_chdir	在循环时不 chdir 到每个目录
untaint	如果打开 -T（污染模式），则不会 cd 到污染的目录
untaint_pattern	应当使用 qr 引用运算符设置，默认设置为 qr|^([-+@\w./]+)$!
untaint_skip	如果设置，就会跳过 *untain_pattern* 失败的目录（子树）。在这种情况下默认为 die
wanted	用于调用 wanted 函数

示例 18.4

```
(UNIX)
1   use File::Find;
2   find(\&wanted, '/httpd', '/ellie/testing' );

3   sub wanted{
        -d $_ && print "$File::Find::name\n";
    }

(Output)
/httpd
/httpd/php
/httpd/Icons
/httpd/Cgi-Win
/httpd/HtDocs
/httpd/HtDocs/docs
/httpd/HtDocs/docs/images
/httpd/Cgi-Bin
/httpd/Logs
/ellie/testing
/ellie/testing/Exten.dir
/ellie/testing/extension
/ellie/testing/mailstuff
/ellie/testing/mailstuff/mailstuff
/ellie/testing/OBJECTS
/ellie/testing/OBJECTS/polymorph
```

解释

1. 从标准 Perl 库中加载 File::Find 模块。
2. find() 的第一个参数是指向子例程 wanted 的引用，后面跟有两个待查找的目录。
3. wanted 函数将检查每个名字是否是目录（-d），并列出所找到目录的完全路径。将 $_ 赋值为正在搜索的当前目录的名字。

示例 18.5

```
(Windows)
1   use File::Find;
```

```
2   use Win32::File;
    # Works on both FAT and NTFS file systems.
3   &File::Find::find(\&wanted,"C:\\httpd", "C:\\ellie\\testing");
4   sub wanted{
5       (Win32::File::GetAttributes($_,$attr)) &&
        ($attr & DIRECTORY) &&
        print "$File::Find::name\n";
    }

(Output)
C:\httpd
C:\httpd/php
C:\httpd/Icons
C:\httpd/Cgi-Win
C:\httpd/HtDocs
C:\httpd/HtDocs/docs
C:\httpd/HtDocs/docs/images
C:\httpd/Cgi-Bin
C:\httpd/Logs
C:\ellie\testing
C:\ellie\testing/Exten.dir
C:\ellie\testing/extension
C:\ellie\testing/mailstuff
C:\ellie\testing/mailstuff/mailstuff
C:\ellie\testing/OBJECTS
C:\ellie\testing/OBJECTS/polymorph
```

解释

1. 从标准 Perl 库中加载 File::Find 模块。
2. 从标准 Perl 库中有关 Win32 系统的站点目录加载 Win32::File 模块，用于检索文件或目录的属性。
3. find() 的第一个参数是指向子例程 wanted 的引用，后面跟有两个待查找的目录。
4. 定义 wanted 函数。
5. 通过调用 GetAttributes 函数（Win32::File::GetAttributes），wanted 函数将检查每个名称是否是目录，并列出所找到的子目录的完全路径。将 $_ 赋值为正在搜索的当前目录的名字。

18.1.4 创建目录——mkdir 函数

UNIX。mkdir 函数能以指定的用户权限（模式）创建新的空目录。用户权限是以一个八进制数字来设置的。系统会自动创建目录项 . 和 ..。如果创建成功，mkdir 会返回 1，否则就返回 0。如果 mkdir 失败的话，还会把系统错误保存到 Perl 的 $! 变量中。

Windows。如果在 MS-DOS 提示符下创建目录，则上述权限掩码（permission mask）将不起任何作用。Win32 上的权限机制不同于 UNIX。对于 FAT 分区上的文件（也就是说所有文件都位于 Windows 95 上），读者无需对文件显式地设置其权限。所有文件都能供所有用户使用，并且创建的目录将对每个用户都打开所有的权限许可。

格式

```
mkdir(FILENAME, MODE); (UNIX)
mkdir(FILENAME);       (Windows)
```

示例 18.6

```
(The Command Line)
1   $ perl -e 'mkdir("joker", 0755);' # UNIX
2   $ ls -ld joker
    drwxr-xr-x  2 ellie          512 Mar    7 13:43 joker
3   $ perl -e "mkdir(joker);"          # Windows
```

解释

1. mkdir 函数的第一个参数是目录名，第二个参数则指定了文件的模式(mode)或权限(permission)。权限 0755 表明文件的拥有者具有读、写和执行权限，其所属的组（group）具有读和执行权限，而其他人则拥有读和执行权限（请记住，如果没有执行权限，就无法访问目录）。
2. ls -ld 命令将打印出一长串目录文件清单，其中包含了文件信息和 inode 信息。其前面的参数 d 表示目录，其权限为 rwxr-xr-x。
3. 在 Win32 系统上，创建的目录将对每个人都开启所有的权限。

示例 18.7

```
    # This script is called "makeit"
1   die "$0 <directory name>  " unless $#ARGV == 0;
2   mkdir ($ARGV[0], 0755 ) || die "mkdir:  $ARGV[0]:  $!\n";

(At The Command Line)
    $ makeit
1   makeit <directory name> at makeit line 3.
    $ makeit joker
2   makeit: joker: File exists
    $ makeit cabinet
    $ ls -d cabinet
    cabinet
```

解释

1. 如果用户没有为脚本提供目录名参数，die 函数就会打印错误消息并退出脚本。
2. 除非目录已经存在，否则创建该目录。

18.1.5 删除目录——rmdir 函数

rmdir 函数负责删除目录，但在删除时该目录必须为空。

格式

```
rmdir(DIRECTORY);
rmdir DIRECTORY;
```

示例 18.8

```
(At the Command Line)
1   $ perl -e 'rmdir("joke") || die qq(joke: $!\n)'    # UNIX
    joke: Directory not empty
2   $ perl -e 'rmdir("joker") || die qq(joker: $!\n)'
    joker: No such file or directory
3   $ perl -e "rmdir(joker) || die qq(joker: $!\n);"   # Windows
    joker: No such file or directory
```

解释

1. 目录 joke 中含有文件。除非目录为空，否则是不能删除它的。$! 变量中含有系统错误信息 Directory not empty。
2. 目录 joke 不存在；因此也不能删除它。系统错误将保存到 $! 中。
3. 在 Win32 系统上，rmdir 的工作方式不变。如果是在 MS-DOS 提示符下，读者只需注意观察引号。这里无法删除 joke 目录，因为它并不存在。

18.1.6 更改目录——chdir 函数

每个进程都具有一个当前工作目录。在解析相对目录引用时，该目录是搜索路径的开始之处。如果调用进程（如 Perl 脚本）更改了目录，则它只对该进程有效，而不能影响调用它的那个进程（一般是 shell）。在退出 Perl 程序时，shell 还会返回开始时的那个工作目录。

chdir 函数能够更改当前的工作目录。它不接受任何参数，能将目录更改到用户的主目录中。如果调用成功，该函数将返回 1，否则就返回 0。系统错误代码将保存在 Perl 的 $! 变量中⊖。

格式

```
chdir (EXPR);
chdir EXPR;
chdir;
```

示例 18.9

```
1   $ pwd        # UNIX
    /home/jody/ellie
2   $ perl -e 'chdir  "/home/jody/ellie/perl"; print 'pwd''
    /home/jody/ellie/perl
3   $ pwd
    /home/jody/ellie
4   $ perl -e 'chdir " fooler" || die "Cannot cd to fooler: $!\n"'
    Cannot cd to fooler: No such file or directory
5   $ cd         # Windows
    C:\ellie\testing
6   $ perl -e "chdir fooler || die qq(Cannot to fooler: $!\n);"
    Cannot cd to fooler: No such file or directory
```

解释

1. 这是 shell 的当前工作目录。
2. 将目录更改为 /home/jody/ellie/Perl。当 pwd 命令位于反引号之间时，执行命令替换操作，打印该进程的当前工作目录。
3. 由于 Perl 程序是 shell 调用的单独进程，如要让 Perl 更改当前工作目录，只有在执行 Perl 进程的过程中才会生效。在退出 Perl 时，重新回到 shell 中，其目录仍将保持不变。
4. 如果更改目录的尝试告以失败的话，die 函数会把本行所示的信息打印到屏幕上。系统错误将保存到 $! 中，然后再打印出来。
5. 在 MS-DOS 提示符下打印当前工作目录。cd 打印当前的工作目录。（在 UNIX 提示符下，该

⊖ chdir 是 Perl 提供的用于更改当前目录的系统调用。命令行中用到的 cd 命令是一种 shell 内建命令，不能用在 Perl 脚本中。

命令也用于更改目录。）

6. 与上述 UNIX 示例相同，更改目录的尝试失败了。如果目录存在，则更改当前工作目录。

18.1.7 通过目录文件句柄访问目录

下面的 Perl 目录函数是模仿同名的 UNIX 系统调用而提供的。尽管传统的 UNIX 目录都含有两个字节的信息节点编号以及 14 个字节的文件名（如图 18-2），但并非所有 UNIX 系统都拥有相同的格式。不论目录的内部结构如何，目录函数都能用于访问目录。这些目录函数在 Windows 中的工作方式也与之相同。

Inode	Filename
10	.
22	..
32	memo
45	mbox
23	notes
12	src

图 18-2　一个 UNIX 目录

opendir 函数。opendir 函数能打开指定的目录，并将它附加到目录文件句柄上。该句柄本身具有自己的命名空间，和那些用于打开文件或筛选器的其他文件句柄不同。opendir 函数负责初始化目录，以便通过与之相关的 readdir()、telldir()、seekdir()、rewinddir() 和 closedir() 函数处理该目录。如果调用成功，该函数将返回 1。

格式

```
opendir(DIRHANDLE, EXPR)
```

示例 18.10

```
opendir(MYDIR, "joker");
```

解释

将文件 joker 附加给文件句柄 MYDIR，并以读模式打开它。目录 joker 必须存在，并且必须是一个目录。

readdir 函数。在目录上拥有读权限的任何人都可以读取该目录。但即使拥有写权限的人也不一定能对目录本身执行写操作。目录上的写权限意味着可以在目录中创建或删除文件，而不是更改目录本身的数据结构。

所谓使用 readdir 函数读目录，其含义是查看由系统维护的目录结构内容。如果 opendir 函数打开了目录，在标量上下文语境中，readdir 就会返回下一个目录项；而在数组上下文语境中，它将返回目录中其余的项。

格式

```
readdir(DIRHANDLE);
readdir DIRHANDLE;
```

closedir 函数。closedir 函数能关闭之前由 opendir 函数打开的目录。

格式

```
closedir (DIRHANDLE);
closedir DIRHANDLE;
```

示例 18.11

```
(The Script)
1   opendir(DIR, "..") || die "Can't open: $!\n";
                         # Open parent directory
2   @parentfiles=readdir(DIR);
                         # Gets a list of the directory contents
3   closedir(DIR);       # Closes the filehandle
4   foreach $file ( @parentfiles )
                         # Prints each element of the array
       { print "$file\n";}

(Output)
.
..
filea
fileb
filec
.sh_history
stories
```

解释

1. opendir 函数负责打开目录结构，并将它附加给目录文件句柄 DIR 上。以读方式打开 .. 目录（即当前目录）。
2. readdir 函数将目录中其余所有的项赋值给数组 @parentfiles。
3. closedir 函数负责关闭目录。
4. 按照在目录结构中的存储顺序打印文件。这可能并不是 ls 命令打印文件的顺序。

telldir 函数。telldir 函数负责返回 readdir() 例程在目录文件句柄上的当前位置。读者可以把 telldir 函数返回的值赋予 seekdir()，以便访问目录中的特定位置。

格式

```
telldir(DIRHANDLE);
```

rewinddir 函数。rewinddir 函数将 DIRHANDLE 的位置设置回到 opendir 所打开目录的开头。并不是所有机器都能支持该函数。

格式

```
rewinddir(DIRHANDLE);
rewinddir DIRHANDLE;
```

seekdir 函数。seekdir 函数会在目录文件句柄上设置 readdir() 的当前位置。用户可使用 telldir() 的返回值来设置该位置。

格式

```
seekdir(DIRHANDLE, POS);
```

示例 18.12

```
(The Script)
```

```
1    opendir(DIR, "."); # Opens the current directory
2    while( $myfile=readdir(DIR) ){
3       $spot=telldir(DIR);
4       if ( "$myfile" eq ".login" ) {
           print "$myfile\n";
           last;
        }
     }
5    rewinddir(DIR);
6    seekdir(DIR, $spot);
7    $myfile=readdir(DIR);
     print "$myfile\n";

(Output)
.login
.cshrc
```

解释

1. opendir 函数以读模式打开当前工作目录。
2. 执行 while 语句。readdir 函数返回文件句柄中的下一项内容，并将文件赋值给标量 $myfile。
3. 在 readdir 函数读取文件名后，telldir 函数标记其读取的位置，并保存到标量 $spot 中。
4. 当读到 .login 文件时，退出循环。
5. rewinddir 函数将 DIR 文件句柄的位置重置为目录结构的开头位置。
6. 利用 telldir 函数的返回结果，seekdir 函数设置了 readdir 函数在 DIR 文件句柄上的当前位置。
7. readdir 函数读取 next 目录项，并赋值给标量 $myfile。

18.1.8 权限和所有权

UNIX。每个 UNIX 文件都有一个文件所有者。该所有者比其他人多出了修改文件权限的权力，因此可以控制其他人对该文件的操作。一个组中可能有很多个成员，文件拥有者也能改变它的组权限，使得这个组可以享有一些特权。

每个 UNIX 文件都有一组权限值与之相关，用于控制谁可以访问、写入或执行该文件。文件权限一共由 9 个比特组成。开头 3 个位负责控制文件所有者的权限；第二组的 3 位负责控制组的权限；最后一组则控制其他人的权限。权限值存储在文件信息节点的模式（model）字段中。

Windows。Win32 系统处理文件权限的方式不同于 UNIX 系统。它创建的文件对每个用户都开启了读写权限。文件和文件夹分别继承了可以设置的属性。读者可以单击文件图标并选择“属性”，然后便可以有限的方式选择权限属性，如 Archive、Read-only 和 Hidden。详见图 18-3。

如果是 Windows NT 平台，则只能在 NTFS 格式的驱动器上设置文件和文件夹的权限⊖。如需更改权限，用户必须是文件所有者，或者拥有其所有者的授权。如果使用的是 NTFS，读者可以进入 Windows Explorer，然后选定要设置权限的文件或文件夹。右击该文件或文件夹，单击“属性”，然后单击“安全”选项卡，便可启用、拒绝或删除针对组或用户的权限。

如需维护文件权限，参阅 Win32 版 Perl Resource Kit 中的 Win32::FileSecurity 模块。若要从文件或目录检索其文件权限，则可使用 Win32::FileSecurity::Get($Path, %Perms) 扩展，其中 $Path 是需要查询权限的文件或目录的相对路径或绝对路径，%perms 是一个散列的引用，其键代表用户和组，值代表相应的权限掩码。表 18-6 显示了管理文件和目录的 Win32 扩展。

⊖ NTFS 是为 Windows NT 设计的高级文件系统。

表 18-6　管理文件和目录的 Win32 扩展

扩　展	用　途
Win32::File	负责检索和设置文件属性的标准模块
Win32::File::GetAttributes(path,attribute)	检索文件属性
Win32::File::SetAttributes(path,attribute)	设置文件属性
Win32::AdminMisc::GetFileInfo	检索文件信息字段：CompanyName、FileVersion、InternalName、LegalCopyright、OriginalFileName、ProductName、ProductVersion、LangID、Language

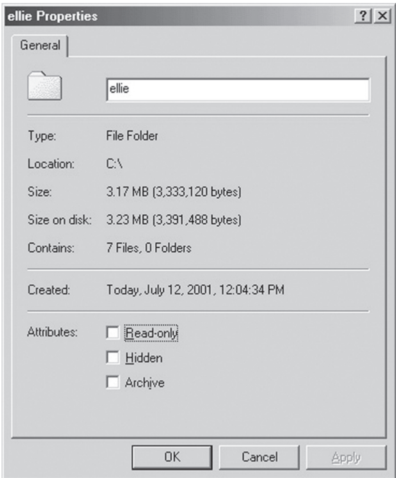


图 18-3　文件属性（Windows）

chmod 函数（UNIX）。chmod 函数负责更改列表或文件的权限。用户必须首先拥有修改这些文件的权限。这些文件必须是带引号的字符串。列表的第一个元素是表示新模式的八进制数字值（目前，二进制 / 八进制表示法已经由更方便记忆的权限更改方法代替了，不过 Perl 并没有使用这种新方法）。

如果读者对这种方法还不熟悉的话，表 18-7 列出了所有八种用于更改权限的数字组合。

表 18-7　权限模式

八进制	二进制	权限	含义
0	000	无	关闭所有权限
1	001	--x	执行
2	010	-w-	写
3	011	-wx	写、执行
4	100	r--	读
5	101	r-x	读、执行
6	110	rw-	读、写
7	111	rwx	读、写、执行

注：请确保其第一位数字为 0，以便表明它是一个八进制数字。请不要使用记忆模式（例如 +rx），因为它会关闭所有的权限。

chmod 函数（Windows）。ActivePerl 只支持有限个版本的 chmod 函数。然而，它只能用于为文件所有者提供读 / 写访问权限（忽略 group 和 other 位）。

chmod 函数将返回经过更改的文件数目。

格式

```
chmod(LIST);
chmod LIST;
```

示例 18.13

```
     (UNIX)
1    $ perl -e '$count=chmod 0755, "foo.p", "boo.p" ;print "$count
            files changed.\n"'
2    2 files changed.
3    $ ls -l foo.p boo.p
     -rwxr-xr-x 1 ellie 0 Mar 7 12:52 boo.p*
     -rwxr-xr-x 1 ellie 0 Mar 7 12:52 foo.p*
```

解释

1. 第一个参数是八进制值 0755，它能为文件所有者开启 rwx 权限，为组和其他人打开 r 和 x 权限。接下来的两个参数 foo.p 和 boo.p 是需要更改的文件。标量 $count 含有经过更改的文件总数。
2. $count 的值为 2。因为上面两个文件的权限都改成了 0755。
3. 打印 UNIX 的 ls -l 命令的输出，表明文件 foo.p 和 boo.p 上的权限都已经改成了 0755。

chown 函数（UNIX）。chown 函数负责改变文件列表的所有者和组。只有文件所有者和超级用户才能调用该函数[1]。列表的前两个元素必须是数字形式的 uid 和 gid。每个授权的 UNIX 用户在口令文件（password file）[2]中都赋予了一个 uid（用户标识号）和一个 gid（组标识号）。该函数将返回经过更改的文件数目。

格式

```
chown(LIST);
chown LIST;
```

示例 18.14

```
(The Script)
1    $ uid=9496;
2    $ gid=40;
3    $number=chown($uid, $gid, 'foo.p', 'boo.p');
4    print "The number of files changed is $number\.n";

(Output)
4    The number of files changed is 2.
```

解释

1. 将用户标识号赋值为 9496。
2. 将组标识号赋值为 40。

[1] 在 BSD UNIX 和某些基于 POSIX 的 UNIX（Solaris）系统中，只有超级用户才能更改所有权。

[2] 如需获得 uid 或 gid，可使用 getpwname 和 getpwuid 函数。

3. chown 函数负责修改 foo.p 和 boo.p 的所有权，并返回经过更改的文件数目。

umaks 函数（UNIX）。在创建一个文件时，该文件默认就拥有一组特定的权限。该权限由所谓的系统掩码（system mask）来决定。在大多数系统上，该掩码都是 022，并由登录程序予以设置[㊀]。在默认情况下，目录会拥有 777（rwxrwxrwx）权限，文件则具有 666（rw-rw-rw-）权限。umask 函数用于在现有的掩码中删除或减去指定的权限。

如要取消“others”的 write 权限，可从每个目录或文件拥有的最大权限值中减去相应的 umask 值：

```
  777 ( 目录 )           666 ( 文件 )
−002 (umask 值 )      −002 (umask 值 )
  775                    664
```

umask 函数负责为本进程设置 umask 值，并返回来的 umask。当没有参数时，umask 函数将返回当前设置情况。

格式

```
umask(EXPR)
umask EXPR
umask
```

示例 18.15

```
1   $ perl -e 'printf("The umask is %o.\n", umask);'
    The umask is 22.
2   $ perl -e 'umask 027; printf("The new mask is %o.\n", umask);'
    The new mask is 27.
```

解释

1. 不带参数的 umask 函数将打印当前的 umask 值。
2. umask 函数将掩码重新设置为八进制值 027。

18.1.9 硬链接和软链接

UNIX。当创建一个文件时，它就拥有一个硬链接；即目录中的一项。用户也可为文件创建额外的链接，该链接只是同一个文件的不同名称而已。内核在文件信息节点中维护了一个值，负责跟踪文件有多少个链接。只要存在指向文件的链接，其数据块就不会对系统释放。让文件具有多个名字的好处在于只需提供一组数据或主文件，并通过很多种不同的名称来访问该文件。硬链接不能跨越文件系统，并且必须在链接创建时出现。

软链接又称为符号链接（symbolic link），有时也称为 symlink。符号链接只是一种很小的文件（具有权限、所有权、大小等信息）。它只含有另一个文件的名称。当访问具有符号链接的文件时，内核会指向符号链接中包含的文件名。例如，从 thisfile 到 /usr/bin/joking/otherfile 的符号链接会把名称 thisfile 链接到 /usr/bin/joking/otherfile 上。当打开 thisfile 时，实际访问的文件则是 otherfile。符号链接可以引用存在或者不存在的文件，而且可以跨越不同的系统，甚至是不同的计算机。它们还可以指向其他的符号链接[㊁]。

㊀ 用户亦可在 .profile（sh 或 ksh）或 .cshrc（csh）初始化文件中设置 umask。

㊁ 符号链接最初出现于 BSD 中，并得到了许多 ATT 系统的支持。不过读者的系统不一定能支持它。

Windows。Win32 系统引用了快捷方式的概念，它是一种扩展名为 .LNK 的特殊二进制文件。快捷方式类似于 UNIX 链接，所不同的是它由特定的应用程序负责处理，而不是由系统自己处理，此外它也是文件或目录的别名。快捷方式是一种图标，其左侧的白框中会出现一个白色箭头。在 Perl 中，读者可以使用 Win32::Shortcut 模块来创建、加载、检索、保存和修改快捷方式的属性（见图 18-4）。

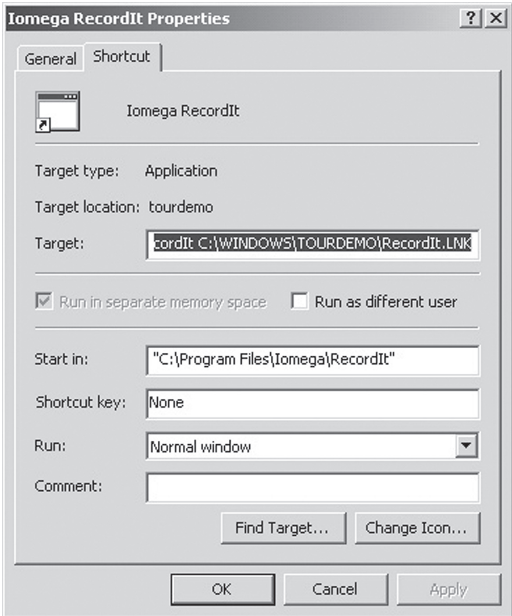


图 18-4　快捷方式和 .LNK 扩展名

link 与 unlink 函数（UNIX）。link 函数能在 UNIX 系统中创建硬链接（即两个具有相同名称的文件）。link 函数的第一个参数是现有的文件名；第二个参数则是新文件名，它必须是不存在的。只有超级用户才能创建指向目录的链接。在删除目录时应当使用 rmdir。

格式

```
link(OLDFILENAME, NEWFILENAME);
```

示例 18.16

```
    (UNIX)
1   $ perl -e 'link("dodo", "newdodo");'
2   $ ls -li dodo newdodo
    142726 -rw-r--r-- 2 ellie          0 Mar 7 13:46 dodo
    142726 -rw-r--r-- 2 ellie          0 Mar 7 13:46 newdodo
```

解释

1. 为旧文件 dodo 提供替代名称 newdodo。
2. ls 命令的 i 选项给出了文件的信息节点号。如果其信息节点号相同，就说明文件是相同的。旧文件 dodo 开始时只拥有 1 个链接。现在链接数从 2 开始计数。因为 dodo 和 newdodo 是链接在一起的，它们是同一个文件，更改其中一个文件就会造成另一个文件的变更。如果删除其中一个链接，则另一个仍然可以存在。如要删除某个文件，首先必须删除其所有的硬链接。

unlink 函数在 UNIX 和 Windows 系统上都能删除文件列表（等效于 UNIX 的 rm 命令和 MS-DOS 中的 del 命令）。如果文件含有多个链接，则将链接计数减去 1。该函数将返回成功删除的文件总数。如需删除目录，则应使用 rmdir 命令，因为只有超级用户才能使用 unlink 函数断开指向目录的链接。

格式

```
unlink (LIST);
unlink LIST;
```

示例 18.17

```
(The Script)
1   unlink('a','b','c') || die "remove: $!\n";
2   $count=unlink <*.c>;
    print "The number of files removed was $count\n";
```

解释

1. 删除文件 a、b 和 c。
2. 删除所有以 .c 结尾的文件（即 C 源文件）。删除的文件数目保存在标量 $count 中。

symlink 和 readlink 函数（UNIX）。symlink 函数负责创建符号链接。如果引用了旧文件名的话，符号链接文件就会访问该文件的名称。

格式

```
symlink(OLDFILE, NEWFILE)
```

示例 18.18

```
1   $ perl -e 'symlink("/home/jody/test/old", "new");'
2   $ ls -ld new
    lrwxrwxrwx  1 ellie 8 Feb 21 17:32 new -> /home/jody/test/old
```

解释

1. symlink 函数创建新文件名 new，并将它链接到旧的文件名 /home/jody/test/old。
2. ls -ld 命令能列出符号链接的文件。符号 -> 指向新的文件名。位于权限前面的 l 表明它是一个符号链接文件。

readlink 函数会返回符号链接的值，如果该文件不是符号链接，则返回未定义。

格式

```
readlink(SYMBOLIC_LINK);
readlink SYMBOLIC_LINK;
```

示例 18.19

```
$ perl -e 'readlink("new")';
/home/jody/test/old
```

解释

文件 new 是一个符号链接，指向 /home/jody/test/old。这也是 readlink 函数所返回的值。

18.1.10 更改文件名

rename 函数（UNIX 和 Windows）。rename 函数负责更改文件的名称，等效于 UNIX 中的 mv 命令。其本质是创建一个指向现有文件的新链接，然后删除原有文件。如果调用成功，rename 函数会返回 1，否则便会返回 0。该命令在工作时不能跨越文件系统的边界。如果已经存在了具有新文件名的文件，则其内容将受到破坏。

格式

```
rename(OLDFILENAME, NEWFILENAME);
```

示例 18.20

```
rename ("tmp", "datafile");
```

解释

将文件 tmp 改名为 datafile。如果 datafile 已经存在，其原有内容将受到破坏。

18.1.11 更改访问与修改时间

utime 函数。utime 函数负责更改文件列表中每个文件的访问和修改时间，类似于 UNIX 中的 touch 命令。列表的头两个元素必须是数字形式的访问时间和修改时间，并且其顺序不能颠倒。time 函数负责为 utime 函数提供当前的时间。该函数将返回成功更改的文件数目。该函数将把列表中每个文件的信息节点（inode）修改时间设置为当前时间。

格式

```
utime (LIST);
utime LIST;
```

示例 18.21

```
(The Script--UNIX)
1   print "What file will you touch (create or change time stamp)? ";
    chop($myfile=<STDIN>);
2   $now=time;  # This example makes the file if it doesn't exist
3   utime( $now, $now, $myfile) || open(TMP,">>$myfile");[a]

(The Command Line)
    $ ls -l brandnewfile
    brandnewfile: No such file or directory

    $ update.p
1  What file will you touch (create or update time stamp) ? brandnewfile

    $ ls -l brandnewfile
2   -rw-r--r--   1  ellie   0 Mar     6  17:13 brandnewfile
```

a. Wall, L., Christianson, T., and Orwant, J., *Programming Perl, 3rd ed.*, O'Reilly & Associates: Sebastopol, CA, 2000。

解释

1. 用户输入文件名，然后更改其访问时间或修改时间；或者当文件不存在时，创建该文件。
2. 将变量 $now 设置为函数 time 的返回值，它是从 UTC 时间 1970 年 1 月 1 日开始的非闰秒数。

3. $now 的第一个参数是访问时间，第二个参数是修改时间，第三个参数则是函数影响到的文件名。如果由于文件不存在而导致 utime 函数调用失败的话，便调用 open 函数创建新文件，并以 TMP 作为文件句柄。该函数类似于 UNIX 中的 touch 命令。

18.1.12 文件统计量

文件信息存储在称为信息节点（inode）的数据结构中，由 系统内核负责维护。对于 UNIX 用户，可使用 ls 命令检索其中很多信息。而在 C 和 Perl 程序中，则可使用 stat 函数直接从信息节点中检索这些信息。见 File::stat 模块，它能为 stat 函数创建用户界面。虽然这里强调是面向 UNIX 的，但在 Win32 系统上也可使用 stat 函数。

stat 和 lstat 函数。 stat 函数会返回一个 13 元素的数组，其中含有从文件信息节点检索得到的统计量。数组中的最后两个字段只与块（block）有关，它们只定义在 BSD UNIX 系统中。[⊖]

lstat 函数与 stat 函数功能相同，但如果文件是一个符号链接，则 lstat 会返回该链接本身的信息，而不是它引用的文件信息。如果系统不支持符号链接，则应使用一般的 stat 函数。

特殊文件局部 underscore 用于从最近一次统计的文件中获取 stat 信息。其返回的 13 元素数组含有保存在 stat 结构中的如下元素（其顺序与 UNIX 系统调用 stat 的略有不同）。

1. 设备号
2. 信息节点号
3. 模式
4. 链接计数
5. 用户 ID
6. 组 ID
7. 对于特殊文件，表示其引用的设备号
8. 以字节（byte）为单位的大小，针对常规文件
9. 上一次访问时间
10. 上一次修改时间
11. 上一次状态更新时间
12. 文件系统首选的 I/O 块大小
13. 分配的 512 字节块的实际数目

格式

```
stat(FILEHANDLE);
stat FILEHANDLE;
stat(EXPR);
```

示例 18.22

```
(UNIX)
open(MYFILE, "perl1") || die "Can't open: $!\n";
@statistics=stat(MYFILE);
print "@statistics\n";
```

⊖ Wall,L.,Christianson,T.,and Orwant,J.,*Programming Perl, 3rd ed.*, O'Reilly & Associates: Sebastopol,CA,2000, p.188。

龍睿 · LoRui
本 PDF 书签由 龙睿 • LoRui 制作。
www.LoRui.com

来看看这些东东……

- ✧ Yahoo!疯了！
- ✧ HP笔记本激活Windows 7

域名转让：

- ✓ DaNanPing.com（大南平）
- ✓ 15500.net
- ✓ Gn7c.com
- ✓ DaLongNan.com（大陇南、大龙南）
- ✓ 51perl.com（无忧 Perl）
- ✓ LoRui.com

```
    close MYFILE;

4   @stats=stat("perl1");
5   printf("The inode number is %d and the uid is %d.\n",
           $stats[1], $stats[4]);
6   print "The file has read and write permissions.\n",
           if -r _ && -w _;

(Output)
3   1819 142441 33261 1 9496 40 -21335 75 761965998 727296409 8192 2
5   The inode number is 142441 and the uid is 9496.
6   The file has read and write permissions.
```

解释

1. 通过文件句柄 MYFILE 打开文件 Perl1。
2. stat 函数从文件的信息节点中检索信息，并将信息返回给 13 元素数组 @statistics。
3. 打印 13 元素数组。数组的最后两个元素是块的大小以及 512 字节块的数目。块的大小和数目可能有所不同，因为在块数中并没有统计尚未分配的块。负值表示 NIS 设备数。
4. 此次 stat 函数以文件名作为参数，而不是文件句柄。
5. 打印数组的第二个和第五个元素。
6. 特殊的下划线（_）文件句柄负责从上一次 stat 调用中检索当前文件的统计信息。文件测试运算符 -r 和 -w 则使用当前的 stat 信息检查该文件的读写权限。

示例 18.23

```
(Windows)
    # Since UNIX and Windows treat files differently,
    # some of the fields here are
    # blank or values returned are not meaningful
1   @stats = stat("C:\\ellie\\testing");
2   print "Device: $stats[0]\n";
3   print "Inode #: $stats[1]\n";
4   print "File mode: $stats[2]\n";
5   print "# Hard links: $stats[3]\n";
6   print "Owner ID: $stats[4]\n";
7   print "Group ID: $stats[5]\n";
8   print "Device ID: $stats[6]\n";
9   print "Total size: $stats[7]\n";
10  print "Last access time: $stats[8]\n";
11  print "Last modify time: $stats[9]\n";
12  print "Last change inode time: $stats[10]\n";
13  print "Block size: $stats[11]\n";
14  print "Number of blocks: $stats[11]\n";

(Output)
2   Device: 2
3   Inode #: 0
4   File mode: 16895
5   # Hard links: 1
6   Owner ID: 0
7   Group ID: 0
8   Device ID: 2
9   Total size: 0
10  Last access time: 981360000
11  Last modify time: 977267374
12  Last change inode time: 977267372
13  Block size:
14  Number of blocks:
```

18.1.13 底层 I/O 文件

read 函数（fread）。 read 函数负责从文件句柄中读取指定数量的字节，并将这些字节放在标量型变量中。读者如果熟悉 C 的标准 I/O 函数 fread 的话，就会发现 Perl 的 read 函数也是以相同方式处理 I/O 缓冲的。为了提高效率，read 函数并不是一次读取一个字节，而是读取一块数据并保存到临时存储区中。然后，C 的 fread 函数与 Perl 的 read 函数会从临时缓冲区将数据一次一个字节地传送给程序。（sysread 函数可用于模拟 C 的底层 I/O 函数 read。）该函数会返回读取的字节总数；或者在发生错误时，返回未定义的数字。如果碰到了 EOF（文件结束符），则返回 0。

在 Perl 中，print 函数（而不是 write 函数）负责输出 read 函数返回的实际字节。Perl 的 print 函数类似于 C 中的 fwrite 函数。

格式

```
read(FILEHANDLE, SCALAR, LENGTH, OFFSET);
read(FILEHANDLE, SCALAR, LENGTH);
```

示例 18.24

```
(The Script)
1   open(PASSWD, "/etc/passwd") || die "Can't open: $!\n";
2   $bytes=read (PASSWD, $buffer, 50);
3   print "The number of bytes read is $bytes.\n";
4   print "The buffer contains: \n$buffer";

(Output)
3   The number of bytes is 50.
4   The buffer contains:
    root:YhTLR4heBdxfw:0:1:Operator:/:/bin/csh
    nobody:
```

解释

1. 通过文件句柄 PASSWD 打开并读取文件 /etc/passwd。
2. read 函数尝试从文件句柄中读取 50 个字节，并将读到的字节数返回给标量 $byte。

sysread 和 syswrite 函数。 sysread 函数类似于 C 的 read 函数。它避开了标准的 I/O 缓冲模式，能直接从文件句柄中读取字节到标量型变量。混用 read 和 sysread 可能会造成问题，因为 read 函数用到了缓冲区，而 sysread 将直接从文件句柄读取字节。

syswrite 函数负责将数据字节写到文件句柄中，它类似于 C 的 write 函数。

格式

```
sysread(FILEHANDLE, SCALAR, LENGTH, OFFSET);
sysread(FILEHANDLE, SCALAR, LENGTH);

syswrite(FILEHANDLE, SCALAR, LENGTH, OFFSET);
syswrite(FILEHANDLE, SCALAR, LENGTH);
```

seek 函数。 Perl 的 seek 函数等效于 C 的标准 I/O 函数 fseek，用于随机访问文件。它会在文件中设定一个位置，并从文件开头以字节为单位表示该位置，其中第一个字节是字节 0。如果调用成功，该函数会返回 1，否则就返回 0。

格式

```
seek(FILEHANDLE, OFFSET, POSITION);

POSITION=The absolute position in the file where
0=Beginning of file
1=Current position in file
2=End of file

OFFSET=Number of bytes from POSITION.A positive offset advances the position forward
 in the file.A negative offset moves the position backward in the file.A negative OFFSET
sets the file position for POSITION 1 or 2.
```

示例 18.25

```
(The Script)
1   open(PASSWD, "/etc/passwd") || die "Can't open: $!\n";
2   while ( chomp($line = <PASSWD>) ){
3      print "---$line---\n" if $line =~ /root/;
    }
4   seek(PASSWD, 0, 0) || die "$!\n"; # Start back at the beginning
                                      # of the file at first byte
5   while(<PASSWD>){print if /ellie/;}
6   close(PASSWD);

(Output)
3   ---root:YhTLR4heBdxfw:0:1:Operator:/:/bin/csh---
5   ellie:aVD17JSsBMyGg:9496:40:Ellie Savage:/home/jodyellie:/bin/csh
```

解释

1. 通过文件句柄 PASSWD 打开文件 /etc/passwd。
2. while 语句循环遍历文件句柄 PASSWD，一次一行地读取其内容，直到文件末尾。
3. 如果文件含有正则表达式 root，则打印该行。
4. seek 函数在文件开头的位置 0 设定文件位置，其字节偏移量（byte offset）为 0。由于这里没有关闭文件句柄，因此它会一直保持打开状态，直到使用 close 函数关闭它。
5. while 语句循环遍历该文件。
6. 正式关闭 PASSWD 文件句柄。

tell 函数。tell 函数能针对常规文件返回其文件句柄的当前位置，该位置可作为调用 seek 函数的参数，以便移动到文件内的某个位置上。

格式

```
tell (FILEHANDLE);
tell FILEHANDLE;
tell;
```

示例 18.26

```
(The Script)
1   open(PASSWD, "/etc/passwd") || die "Can't open: $!\n";
    while ( chomp($line = <PASSWD>) ){
        if ( $line =~ /sync/){
2          $current = tell;
           print "---$line---\n";
        }
    }
```

```
3   printf "The position returned by tell is %d.\n", $current;
4   seek(PASSWD, $current, 0);
    while(<PASSWD>){
5       print;
    }

(Output)
2   --sync::1:1::/:/bin/sync--
3   The position returned by tell is 296.
5   sysdiag:*:0:1:Old System Diagnostic:/usr/diag/sysdiag/sysdiag
    sundiag:*:0:1:System Diagnostic:/usr/diag/sundiag/sundiag
    ellie:aVD17JSsBMyGg:9496:40:Ellie Savage:/home/jodyellie:/bin/csh
```

解释

1. 通过文件句柄 PASSWD 打开文件 /etc/passwd。
2. 如果到达了含有正则表达式 sync 的行，由 tell 函数将当前字节位置返回给标量 $current。当前的位置是上次读取的字节的下一个字节。
3. 打印 tell 函数返回的字节位置。
4. seek 函数从文件开头定位到 tell 函数所返回的当前位置的偏移量上。

18.1.14 压缩和解压缩数据

读者还记得 printf 和 sprintf 函数吗？它们能把参数格式化为浮点数、十进制整数、字符串等形式。pack 和 unpack 函数则更进了一步。这两个函数能把字符串表示成比特、字节、整数、长整数、浮点数等形式。格式类型负责告诉 pack 和 unpack 函数应当如何处理这些字符串。

pack 和 unpack 函数具有多种用途。这两个函数可以用于将列表压缩成二进制结构，再从压缩值重新展开成列表。当处理文件时，读者可使用这些函数创建 uuencode 编码文件、关系型数据库和二进制文件。

在处理多个文件时，并非所有的文件都是文本文件。例如，有些文件可能会压缩成某种二进制形式，以便节省空间、存储图像；或者可压缩为 uuencode 格式，以便通过电子邮件传递文件。这些文件都不是可读的。pack 和 unpack 函数还可用于转换文件中多行内容的格式。pack 函数可以把列表转换为保存在计算机内存中的标量值。表 18-8 中列出的模板用于规定字符类型以及需要格式化多少字符。例如，字符串 c4 或 cccc 表示把列表压缩为 4 个无符号字符；a14 表示将列表压缩为 14 个字节的 ASCII 字符，并填充 null 值。unpack 函数能将二进制格式的字符串转换为列表。它是 pack 的逆操作，能将字符串转换回 Perl 格式。

表 18-8 模板 pack 和 unpack：类型和值

模板	描 述
a	ASCII 字符串（填充空值）
A	ASCII 字符串（填充空格）
b	位字符串（从低位到高位顺序，与 vec 相同）
B	位字符串（从高位到低位顺序）
c	带符号的 char 值
C	无符号的 char 值

（续）

模板	描　述
d	本地格式的双精度浮点数
f	本地格式的单精度浮点数
h	十六进制字符串（低四位在前）
H	十六进制字符串（高四位在前）
i	带符号的整数
I	无符号的整数
l	带符号的长整数
L	无符号的长整数
n	“network”（高位优先）顺序的短整数
N	“network”（高位优先）顺序的长整数
p	一个指针，指向一个以空值结束的字符串
P	一个指针，指向一个结构（长度固定的字符串）
q	带符号的 64 位值
Q	无符号的 64 位值
s	带符号的短值（16 位）
S	无符号的短值（16 位）
u	uuencode 编码字符串
v	“VAX”（低位优先）顺序的短整数
V	“VAX”（低位优先）顺序的长整数
w	以 128 为基数的 BER 压缩的无符号整数，高位在先
x	空字节
X	备份字节
@	空值填入绝对位置

格式

```
$string=pack(Template, @list );
@list = unpack(Template, $string );
```

示例 18.27

```
(The Script)
1   $bytes=pack("c5", 80,101,114, 108, 012); # 5 ASCII characters
2   print "$bytes\n";

(Output)
Perl
```

解释

1. 列表中的第一个元素是模板（见表 18.8），它由类型和压缩值的数目组成；在本例中，它是四个带符号的字符。列表的其余部分由字符 P、e、r 和 l 的十进制值构成。将该列表压缩为二进制结构。返回含有压缩结构的字符串，并将它保存到 $bytes 中。（参阅 ASCII 表。）
2. 打印四个字节的字符串。

示例 18.28

```
(Script)
1   $string=pack("A15A3", "hey","you"); # ASCII string, space padded
2   print "$string";

(Output)
2   hey            you
```

解释

1. 使用模板 A15A3 将两个字符串 hey 与 you 压缩为结构。A15 表明把字符串 hey 转换为由 15 个字符组成的 ASCII 字符串，并填充空格。A3 表示把字符串转换为由 3 个字符组成的字符串，并填充空格。
2. 根据 pack 函数的格式模板打印字符串内容。它们都是左对齐的。

示例 18.29

```
(The Script)
    #!/bin/perl
    # Program to uuencode a file and then uudecode it
1   open(PW, "/etc/passwd") || die "Can't open: $!\n";
2   open(CODEDPW, ">codedpw") || die "Can't open: $!\n";

3   while(<PW>){
4       $uuline=pack("u*", $_); # uuencoded string
5       print CODEDPW $uuline;
    }
    close PW;
    close CODEDPW;

6   open(UUPW, "codedpw") || die "Can't open: $!\n";
    while(<UUPW>){
7       print;
    }
    close UUPW;
    print "\n\n";

8   open(DECODED, "codedpw") || die;
9   while(<DECODED>){
10      @decodeline = unpack("u*", $_);
11      print "@decodeline";
    }

(Output)
7   E<F]O=#IX.C`Z,3I3=7!E<BU5<V5R.B\Z+W5S<B]B:6XO8W-H"@``
    19&%E;6]N.G@Z,3HQ.CHO.@H`
    58FEN.G@Z,CHR.CHO=7-R+V)I;CH*
    .<WES.G@Z,SHS.CHO.@H`
    :861M.G@Z-#HT.D%D;6EN.B]V87(O861M.@H`
    L;'`Z>#HW,3HX.DQI;F4@4')I;G1E<B!!9&UI;CHO=7-R+W-P;V]L+VQP.@H`
```

```
    ?<VUT<#IX.C`Z,#I-86EL($1A96UO;B!5<V5R.B\Z"@!R
    E=75C<#IX.C4Z-3IU=6-P($%D;6EN.B]U<W(O;&EB+W5U8W`Z"@!L
    M;G5U8W`Z>#HY.CDZ=75C<"!!9&UI;CHO=F%R+W-
    P;V]L+W5U8W!P=6)L:6,Z
    5+W5S<B]L:6(O=75C<"]U=6-I8V\*
    J;&ES=&5N.G@Z,S<Z-#I.971W;W)K($%D;6EN.B]U<W(O;F5T+VYL<SH*
    ?;F]B;V1Y.G@Z-C`P,#$Z-C`P,#$Z3F]B;V1Y.B\Z"@`O
    I;F]A8V-E<W,Z>#HV,#`P,CHV,#`P,CI.;R!!8V-E<W,@57-E<CHO.@H`
    J;F]B;V1Y-#IX.C8U-3,T.C8U-3,T.E-U;D]3(#0N>"!.;V)O9'DZ+SH*
    M96QL:64Z>#HY-
    #DV.C0P.D5L;&EE(%%U:6=L97DZ+VAO;64O96QL:64Z+W5S
    *<B]B:6XO8W-H"@!C

11  root:x:0:1:Super-User:/:/usr/bin/csh
    daemon:x:1:1::/:
    bin:x:2:2::/usr/bin:
    sys:x:3:3::/:
    adm:x:4:4:Admin:/var/adm:
    lp:x:71:8:Line Printer Admin:/usr/spool/lp:
    smtp:x:0:0:Mail Daemon User:/:
    uucp:x:5:5:uucp Admin:/usr/lib/uucp:
    nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
    listen:x:37:4:Network Admin:/usr/net/nls:
    nobody:x:60001:60001:Nobody:/:
    noaccess:x:60002:60002:No Access User:/:
    nobody4:x:65534:65534:SunOS 4.x Nobody:/:
    ellie:x:9496:40:Ellie Quigley:/home/ellie:/usr/bin/csh
```

解释

1. 以读方式打开本地文件 passwd。
2. 以写方式打开另一个文件 codepw。
3. 将文件句柄的每一行读取 $_，直到文件末尾。
4. pack 函数对该行内容($_)进行 uuencode 编码，并将编码后的行赋予标量 $uuline。uuencode 编码通常用于将二进制文件转换为可供电子邮件发送的编码格式。
5. 将 uuencode 编码字符串发送到文件句柄。
6. 以读方式打开含有 uuencode 编码文本的文件，并将它附加给文件句柄 UUPW。
7. 打印每一行 uuencode 编码文本内容。
8. 以读方式打开 uuencode 编码文件。
9. 从文件句柄中读取每一行内容，并保存到 $_ 中。
10. unpack 函数负责将 uuencode 编码字符串转换回原来的格式，并赋值给 @decodeline。
11. 打印 uuencode 编码行。

示例 18.30

```
(The Script)
    #!/bin/perl
1   $ints=pack("i3", 5,-10,15);    # pack into binary structure
2   open(BINARY, "+>binary" ) || die;
3   print BINARY "$ints";
4   seek(BINARY, 0,0) || die;
    while(<BINARY>){
5       ($n1,$n2,$n3)=unpack("i3", $_);
```

```
6       print "$n1 $n2 $n3\n";
    }

(Output)
6   5 -10 15
```

解释

1. 将三个整数 5、10、−5 压缩为三个带符号的整数。返回值是一个二进制结构。将它赋值给 $ints。
2. 以读写方式打开文件句柄 BINARY。
3. 将压缩得到的整数发送到文件中。该文件是一个压缩文件，并且完全是不可读的。如要读取该文件，读者必须先将其转换回 ASCII 格式。可使用 unpack 函数完成该项任务。
4. seek 函数将文件指针置于文件顶部的字节 0 处。
5. 一次从文件中读取一行内容。解压 $_ 中的每一行内容，并返回原先的列表值。
6. 打印原先的列表值。

示例 18.31

```
(The Script)
    $str="0x123456789ABCDE ellie...";
1    print "$str\n";

    $bytes=unpack("H*",$str); # hex string (regular order)
2    print "$bytes\n";

    $str2 = pack("H*", $bytes);
3    print "$str2\n";

    $bytes = unpack("h*",$str); # hex string (reversed order)

4    print "$bytes\n";

    $str1 = pack("h*", $bytes);
5    print"$str1\n";
(Output)
1  0x123456789ABCDE ellie...
2  3078313233343536373839414243444520656c6c69652e2e2e
3  0x123456789ABCDE ellie...
4  038713233343536373839314243444540256c6c69656e2e2e2
5  0x123456789ABCDE ellie...
```

解释

1. 该字符串含有一个十六进制整数和一些文本。
2. “h”和“H”字段表明对长度为多个 nybble（四位组，表示一个十六进制数，即 0-9a-f）的字符串进行压缩。pack() 函数输入的每个字节都将在结果中生成一个 4 位组。变量 $bytes 中的十六进制字符串以常规十六进制顺序排列而成，原字符串中的每个字符都会表示为两个十六进制数字。例如，“ellie”表示为 65 6c 6c 69 65，而三个点号则可表示为 e2 e2 e2。

18.2 进程

Perl 脚本是驻留在磁盘上的程序。当把一个程序载入内存并开始执行它时，就称它为进程。每个进程都拥有很多从父进程（又称调用进程）继承而来的属性。Perl 提供了一些用于检索进程信息

的函数。在介绍这些函数之前，为帮助读者理解这些 Perl 系统调用的目的，首先简要地介绍一下有关进程的知识。

18.2.1 UNIX 进程

每个进程都有惟一的进程 ID，它是一个正整数，又称作 pid。除了进程 0（即交换器，swapper）之外，每个进程都有自己的父进程。第一个进程是 init，其 pid 为 1，是所有未来进程的共同祖先。后来出现的进程称为其后代（descendant），通常又称为子进程（child process）。

在图 18-5 中，Perl 进程是 shell（sh）的后代。

每个进程也都属于某个进程组（process group）。进程组是用于作业控制和信号处理的进程集合。每个进程组也都拥有惟一的 pid，并有一个进程领导（process leader）。在登录时，进程组领导可以是登录 shell。从 shell 创建的任何进程都是相应进程组的成员。进程组领导打开的终端又称作控制终端（controlling terminnal），从它产生的任何进程都会继承它。任何发送给进程组的信号都会发送到组中所有的进程上。这就是为什么按下 <Ctrl>-c 时，正在运行的进程及其子程序都会终止。Perl 提供了负责获取进程组 ID 和设置进程组的函数。

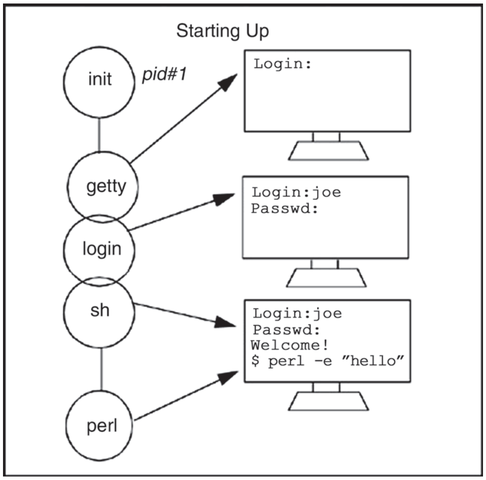


图 18-5　Perl 进程是 shell 的一个后代

在创建进程时，应当为其指定四个数字，以表明进程的所有者。它们必须是真实有效的用户 ID 和真实有效的组 ID。用户 ID 又称 uid，是一个与登录用户名相关联的正整数。真实 uid 是 /etc/passwd 文件中的第三个字段。在登录时，创建的第一个进程又称作登录 shell（login shell），并为它指定用户 ID。从 shell 产生的任何进程都将继承这个 uid。任何以该 uid 运行的进程都称为根进程（root process）或超级用户进程，具有特殊的权限。

此外还有一个组 ID 号，又称真实的 gid，负责把组关联到登录用户名上。默认的 gid 是口令文件中的第四个字段，任何子程序也都会继承它。系统管理员可以通过指定 /etc/passwd 文件中的设置项来允许某个用户成为其他组的成员。

下面是 passwd 文件中的一项，说明了它是如何存储 uid 和 gid 值（字段之间由冒号隔开）。

示例 18.32

```
(Entry from /etc/passwd)
    john:aYD17IsSjBMyGg:9495:41:John Doe:/home/dolphin/john:/bin/ksh
```

解释

（字段）

1. 登录名
2. 加密后的口令
3. uid
4. gid
5. gcos
6. home 目录
7. 登录 shell

通常会把进程的有效 uid（euid）和有效 gid（egid）设置为与运行进程的用户的真实 uid 和 gid 相同。UNIX 会根据 uid 和 gid 决定进程使用的权限。如果文件的 euid 和 egid 变成其他所有者，则该用户会在执行程序时成为进程的实际所有者，并获得其访问权限。负责设置有效 uid 和有效 gid 的程序称作用户 ID 设置程序，又称 setuid 程序。在更改口令时，/etc/passwd 程序将 setuid 为 root 身份，从而为用户提供在 passwd 文件中更改口令的特权，该特权由 root 拥有。

18.2.2　Win32 进程

Windows 的进程模型不同于 UNIX 系统。由于 Perl 原本是为 UNIX 系统设计的，因此标准 Perl 库添加了很多库例程，以便适应 Windows 环境。Win32 目录（C:/Perl/lib/Win32）是一个专用于 Windows 的目录，它随 Perl 的 Windows 版发布，其中含有很多模块，可用于创建、挂起、恢复和撤销进程。Win32::Process 模块中含有很多能够操纵进程的函数。下面列出的是 Win32 目录中的模块清单：

```
AuthenticateUser.pm   Internet.pm        Registry.pm
ChangeNotify.pm       Mutex.pm           Semaphore.pm
Client.pl             NetAdmin.pm        Server.pl
Clipboard.pm          NetResource.pm     Service.pm
Console.pm            ODBC.pm            Shortcut.pm
Event.pm              OLE                Sound.pm
EventLog.pm           OLE.pm             Test.pl
File.pm               PerfLib.pm         TieRegistry.pm
FileSecurity.pm       Pipe.pm            WinError.pm
IPC.pm                Process.pm         test-async.pl
```

18.2.3　环境（UNIX 和 Windows）

在登录时，shell 程序会继承一组环境变量，后者由其他登录程序或某个 shell 启动文件（.profile 或 .login）负责初始化。在这些变量中含有有关进程的有用信息，譬如搜索路径、主目录、用户名和终端类型等。在设置并导出环境变量后，该进程孵化的任何子进程都能继承它们。shell 进程会把环境变量信息传送给 Perl 程序。

特殊关联数组 %ENV 含有一些环境变量方面的设置。如果改变了 Perl 脚本中的环境变量设置

值，则会影响该进程及其所有子进程的设置。而父进程（一般是 shell）中的环境变量将保持不变。

示例 18.33

```
(The Script)
(UNIX)
1   foreach $key (keys(%ENV)){
2          print "$key\n";}
3   print "Your login name is $ENV{'LOGNAME'}\n";
4   $pwd=$ENV{'PWD'};
    print "/The present working directory is $pwd, "\n";

(Output)
2   OPENWINHOME
    MANPATH
    FONTPATH
    LOGNAME
    USER
    TERMCAP
    TERM
    SHELL
    PWD
    HOME
    PATH
    WINDOW_PARENT
    WMGR_ENV_PLACEHOLDER
3   Your login name is ellie
4   The present working directory is /home/jody/ellie
```

解释

1. 使用 keys 函数从 %ENV 数组中获取当前设置的所有环境变量。这些变量都继承自其父进程 shell。
2. 打印每个环境变量。
3. 打印 LOGNAME 的值，即用户名。
4. 将 PWD 的值（即当前工作目录）赋值给 $pwd，并打印之。

示例 18.34

```
(Windows)
    while(($key,$value)=each(%ENV)){
        print "$key: $value\n" if $key =~ /^P/;
    }

(Output)
1   PROMPT: $p$g
    PROCESSOR_IDENTIFIER: x86 Family 6 Model 5 Stepping 2,
        GenuineIntel
    PATH: c:\Oracle\Ora81\bin;"C:\PROGRAM FILES
        \OCLE\JRE\1.1.7\BIN";C:\ORAWIN95\BIN;C:\PERL\BIN\;
        C:\MKSNT;C:\WINDOWS;C:\WINDOW
    PATHEXT: .pl;.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
    PROGRAMFILES: C:\PROGRA~1
    PROCESSOR_ARCHITECTURE: x86
    PROCESSOR_REVISION: 0502
    PERL5LIB: C:\Perl\lib
```

```
    PROCESSOR_LEVEL: 6
```

解释

使用 each 函数从 %ENV 数组中获得当前设置的所有环境变量。这些变量都继承自其父进程 MS-DOS shell。

18.2.4 进程和文件句柄

正如在第 10 章“获得文件句柄”所讨论的，在 Perl 中可以通过输入或输出文件句柄来打开进程。例如，如要查看机器上正在运行的所有进程，可以为 UNIX 的 ps 命令创建一个文件句柄（详见第 10 章，另见本章“system 函数”一节）。

示例 18.35

```
      (The Script)
          # UNIX ps command
      1   open(PROC, "ps -aux |" ) || die "$!\n";
                         # If running System V, use ps -ef
      2   print STDOUT <PROC>;

      (Output)
2  ellie  3825  6.4  4.5  212  464 p5  R 12:18  0:00 ps -aux
   root      1  0.0  0.0   52    0 ?  IW Feb 5  0:02 /sbin/init
   root     51 10.0  0.0   52    0 ?  IW Feb 5  0:02 portmap
   root      2 10.0  0.0   52    0 ?   D Feb 5  0:02 pagedaemon
   root     90 10.0  0.0   52    0 ?  IW Feb 5  0:02 rpc.statd
                             <more processes here>

   ellie  1383  0.8  8.4  360  876 p4  S Dec 26 11:34 /usr/local/OW3/bin/xview
   ellie   173  0.8 13.4 1932 1392 co  S Dec 20389:19 /usr/local/OW3/bin/xnews
   ellie   164  0.0  0.0  100    0 co IW Dec 20  0:00 -c

                   <some of the output was cut to save space>

   ellie  3822  0.0  0.0    0    0 p5  Z Dec 20 0:00 <defunct>
   ellie  3823  0.0  1.1   28  112 p5  S 12:18  0:00 sh -c ps -aux | grep '^'
   ellie  3821  0.0  5.6  144  580 p5  S 12:18  0:00 /bin/perl checkon ellie
   ellie  3824  0.0  1.8   32  192 p5  S 12:18  0:00 grep ^ellie
```

解释

1. 以读方式打开文件句柄 PROC。它又叫作输入过滤器（input filter）。将 ps 命令输出的内容通过 PROC 文件句柄发送给 Perl。
2. 将过滤器的内容打印到 STDOUT 中。

登录信息——getlogin 函数。getlogin 函数负责从 /etc/utmp 中返回当前的登录信息。如果 getlogin 函数返回 null，则可使用 getpwuid。getpwuid 函数以用户的 uid 作为参数，能够从口令文件中返回该 uid 的关联项。

$< 变量是该进程的真实 uid。

格式

```
getlogin;
```

示例 18.36

```
(The Script)
1   $loginname=getlogin || (getpwuid($<))[0]|| die "Not a user here!!";
2   print "Your loginname is $loginname.\n";

(Output)
2   Your loginname is john.
```

解释

1. getlogin 函数从 /etc/utmp 中返回登录名。如果调用失败，则使用 getpwuid 函数检索口令文件。变量 $< 含有该进程的真实 uid。
2. 标量 $loginname 含有用户的登录名，即口令文件的第一项内容。

特殊进程变量（pid、uid、euid、gid、egid）。Perl 提供了一些特殊字符，负责存储执行脚本的 Perl 进程信息。如果要让程序易于阅读，可使用标准 Perl 库中的 English 模块以英语形式表达这些变量。

$$	运行脚本的 Perl 程序的进程 ID
$<	进程的实际 uid
$>	进程的有效 uid
$(	进程的实际 gid
$)	进程的有效 gid

父进程 ID——getppid 函数和 $$ 变量。系统上的每一个进程都由其进程标识号（pid）予以标识，该进程标识号是一个正整数。特殊变量 $$ 中含有该进程 pid 的值。shell 也使用这个变量来维护当前进程的进程 ID 号。

getppid 函数会返回父进程的进程 ID 号。

示例 18.37

```
(The Script)
1   print "The pid of this process is $$\n";
2   print "The parent pid of this process is ", getppid,"\n";

(Output)
1   The pid of this process is 3304
2   The parent pid of this process is 2340

(At the Command Line)
3   $ echo $$
    2340
```

解释

1. 打印该进程（即该 Perl 脚本）的进程标识号（pid）。
2. 原来的 shell 是孵化该进程的进程，也就是它的父进程。父进程的 pid 为 ppid。
3. 退出 Perl 脚本后，使用 $$ 打印 shell 的 pid。Perl 脚本的 ppid 是 2340，也就是其父进程（即 shell）的 pid。

进程组 ID——pgrp 函数。pgrp 函数返回指定 pid 的当前组进程。如果没有参数或者以 0 作为参数的话，该函数将返回当前进程的进程组 ID。

格式

```
getpgrp(PID);
getpgrp PID;
getpgrp;
```

示例 18.38

```
(The Script)
1   print "The pid of the Perl program running this script is ", $$;
2   printf "The ppid, parent's pid (Shell) , is %d\n", getppid;
3   printf "The process group's pid is %d\n", getpgrp(0);

(Output)
1   The pid of the Perl program running this script is 6671
2   The ppid, parent's pid (Shell), is 6344
3   The process group's pid is 6671
```

18.2.5 进程优先级和 Nice 值

内核为每个进程维护着一个选定的调度优先级。大多数具有交互性的短作业都具有高优先级。UNIX 的 nice 命令允许用户修改进程（BSD、pre-System V）的调度优先级。当系统负载较重时，应当让 CPU 密集型作业暂缓执行，以便较高优先级的作业能够更快地访问到 CPU。不贪婪使用处理器的作业又称作是 nice 的。

nice 值用于计算进程的优先级。nice 值为正数的进程将运行在较低的优先级上，意味着其收到的 CPU 时间要少于它的 CPU 时间份额。nice 值为负数的进程将运行在较高的优先级上，其收到的 CPU 时间要多余它的 CPU 时间份额。nice 值的范围是从 −20 到 19。大多数进程都运行在优先级 0 上，以便平衡它们对 CPU 的访问。（只有超级用户才能将 nice 值设置为负数。）

下面的 getpriority 和 setpriority 函数是根据相应的系统调用而命名的，读者可以在 UNIX man 页面的第二部分中找到它们。

getpriority 函数。getpriority 函数能返回进程、进程组或者用户的当前优先级（nice 值）。并不是所有的系统都能支持该函数。如果不能实现，该函数会产生致命错误。WHICH 有三种值：0 表示进程优先级，1 表示进程组优先级，2 则表示用户的优先级。WHO 的解释是与进程优先级、进程组优先级和用户优先级的进程标识符相关联的。值 0 表示是当前进程、进程组或用户。

格式

```
getpriority(WHICH, WHO);
```

示例 18.39

```
(The Script)
1   $niceval = getpriority( 0,0);
2   print "The priority, nice value, for this process is $niceval\n";

(Output)
2   The priority, nice value, for this process is 0.
```

解释

1. getpriority 函数返回当前进程的 nice 值。
2. 该进程的 nice 值为 0。该进程在享用 CPU 时间方面没有优先权。

setpriority 函数。setpriority 函数负责对进程、进程组或用户设置当前的优先级，并修改进程的调度优先级。如果在系统上不能执行 setpriority 系统调用，则 setpriority 函数将返回错误。

WHICH 有三个可选值：0 表示进程优先级，1 表示进程组优先级，2 则表示用户的优先级。WHO 的解释是与进程优先级、进程组优先级和用户优先级的进程标识符相关联的。值 0 表示是当前进程、进程组或用户。NICEVALUE 表示 nice 值。较低的 nice 值能够提高进程的优先级，而较高的 nice 值则能降低进程的优先级。

除非拥有超级用户特权，否则就不能设置负的 nice 值。如果强行这么做，则当前的 nice 值将不会改变。

格式

```
setpriority(WHICH, WHO, NICEVALUE);
```

示例 18.40

```
(The Script)
1   $niceval = getpriority(0,0);
2   print "The nice value for this process is $niceval.\n";
3   setpriority(0,0, ( $niceval + 5 ));
4   print "The nice value for this process is now", getpriority(0,0);

(Output)
2   The nice value for this process is 0.
4   The nice value for this process is now 5.
```

解释

1. getpriority 函数返回当前进程的 nice 值。
2. 打印 nice 值。
3. setpriority 函数将当前进程的 nice 值加上 5。该进程将拥有更低的优先权，也就是更为“nice”。
4. getpriority 函数返回的 nice 值为 5。

18.2.6 口令信息

UNIX。下列函数能迭代遍历 /etc/passwd 文件，并从该文件中检索信息。这些函数的命名方式和系统库中的函数相同（参阅 UNIX 手册的第三部分），并能完成相同的任务。如果读者对有关 /etc/group 文件的信息感兴趣，不妨使用 Perl 函数 getgrent、getgrgid 和 getgrnam，它们都能返回含有 4 个元素的数组，其中含有组（gourp）项目的相关信息。读者可以在 UNIX 手册页中找到这些函数的描述信息。下面是 /etc/passwd 文件示例：

```
root:YhTLR4heBdxfw:0:1:Operator:/:/bin/csh
nobody:*:65534:65534::/:
sys:*:2:2::/:/bin/csh
bin:*:3:3::/bin
uucp:*:4:8::/var/spool/uucppublic:
news:*:6:6::/var/spool/news:/bin/csh
sync::1:1::/:/bin/sync
ellie:aVD17TSsBMfYg:9496:40:Ellie Shellie:/home/jody/ellie:/bin/ksh
```

Windows。Windows2000 和 NT 把有关用户的信息都存放在一个二进制数据库 SAM（Security Accounts Manager）中。该数据库是注册表的一部分。由于它是以二进制形式存储的，因此一般的

Perl 读操作将不起作用。最好使用 Win32 扩展来获取用户信息。Win32::NetAdmin 在 \Perl\site\lib\win32 下与 ActiveState 绑定在一起（详见表 18-9）。读者可使用该模块的两个函数来操纵用户账号：UserGetAttributes() 和 UserSetAttributes()。另一种好的扩展是 David Roth 的 Win32::AdminMisc，可在 www.roth.net 处获得。

Win32 的 net.exe 命令也能显示有关用户与系统的信息。

示例 18.41

```
1   C:\  net help
    The syntax of this command is:

    NET HELP command
       -or-
    NET command /HELP

    Commands available are:

    NET ACCOUNTS               NET HELP          NET SHARE
    NET COMPUTER               NET HC:\ELPMSG    NET START
    NET CONFIG                 NET LOCALGROUP    NET STATISTICS
    NET CONFIG SERVER          NET NAME          NET STOP
    NET CONFIG WORKSTATION     NET PAUSE         NET TIME
    NET CONTINUE               NET PRINT         NET USE
    NET FILE                   NET SEND          NET USER
    NET GROUP                  NET SESSION       NET VIEW
    NET HELP SERVICES lists the network services you can start.
    NET HELP SYNTAX explains how to read NET HELP syntax lines.
    NET HELP command | MORE displays Help one screen at a time.

2   C:\ net user

    User accounts for \\HOMEBOUND

    -------------------------------------------------------------------

    Administrator              Ellie Quigley            Guest
    SQLAgentCmdExec
    The command completed successfully.
```

表 18-9　Win32::NetAdmin 扩展

Win32::NetAdmin:: UserhetAttributes(	*$Machine,*
	$UserName,
	$Password,
	$PasswordAge,
	$Privilege,
	$Homedir,
	$Comment,
	$Flags,
	$ScriptPath);

（续）

Win32::NetAdmin:: UserSetAttributes(*$Machine,* *$UserName,* *$Password,* *$PasswordAge,* *$Privilege,* *$Homedir,* *$Comment,* *$Flags,* *$ScriptPath);*

示例 18.42

```
use Win32::NetAdmin qw(GetUsers UserGetAttributes) ;
GetUsers("", FILTER_NORMAL_ACCOUNT,\%hash)or die;
foreach $key(sort keys %hash){
   print "$key\n";
}
(Output)
Administrator
Ellie Quigley
Guest
SQLAgentCmdExec
```

加密的口令不能从 UNIX 传送到 Win32 系统，反之也不能。它们的加密方式是不兼容的。如需管理口令，应当使用 Win32::AdminMisc 或 Win32::NetAdmin 模块扩展。详见表 18-10。

表 18-10　Win32 口令扩展

Win32::AdminMisc::UserCheckPassword($Machine, $User, $Password);
Win32::AdminMisc::SetPassword(($Machine\|$Domain), $User, $NewPassword);
Win32::AdminMisc::UserChangePassword(($Machine\|$Domain),$User,$OldPassword,$NewPassword);
Win32::NetAdmin::UserChangePassword(($Machine\|$Domain),$User,$OldPassword,$NewPassword);

对于 Windows 用户，下列这几个函数能获得无法从 ActiveState 中实现的组和用户信息：

```
endgrent(), endpwent(), getgrent(), getgrgid(), getgrnam(),
getpwent(), getpwnam(), getpwuid(), setgrent(), setpwent()
```

获取口令项（UNIX）——getpwent 函数。 getpwent 函数能从 /etc/passwd 文件中检索信息。getpwent 函数的返回值是一个 9 元素数组，由如下内容组成：

1. 登录名
2. 加密后的口令
3. 用户 ID
4. 组 ID
5. 配额（Quota）

6. 注释
7. Gcos（用户信息）
8. 主目录
9. 登录的 shell

格式

```
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir,
      $shell )=getpwent;
```

示例 18.43

```
(The Script)
1   while( @info=getpwent) {
2        print "$info[0] \n" if $info[1]= ~/ \*+/;
    }

(Output)
2   nobody
    daemon
    sys
    bin
    uucp
```

解释

1. getpwent 函数从 /etc/passwd 文件中获得一行内容，并保存到数组 @info 中。循环持续执行，直到 getpwent 无法从 /etc/passwd 中读取项目为止。
2. 如果数组的第二个元素含有至少一个星号（*），则打印其第一个元素，即用户名。

通过用户名获得口令项——getpwnam 函数。getpwnam 函数以用户名作为参数，并返回一个 9 元素数组，该数组对应于 /etc/passwd 文件中的用户名。

格式

```
getpwnam(loginname);
```

示例 18.44

```
(The Script)
    #!/bin/perl
1   foreach $name ( "root", "bin", "ellie" ){
2        if (($login, $passwd, $uid)=getpwnam($name)){
3            print "$login--$uid\n";
         }
    }

(Output)
3   root--0
    ellie--9496
    bin--3
```

解释

1. foreach 循环列表中含有一组范围从 1 到 10 的 uid 号，依次处理其中每一个号码。

2. getpwnam 函数从 /etc/passwd 检索信息，并将信息的前三个字段分别存储在数组元素 $login、$passwd 和 $uid 中。
3. 打印登录名及其相应的 uid。

通过 uid 获得口令项——getpwuid 函数。getpwuid 函数以用户的 ID 号作为参数，并返回一个 9 元素数组，该数组对应于 /etc/passwd 文件中的用户 uid 项。

格式

```
getpuid(UID)
```

示例 18.45

```
(The Script)
1   foreach $num ( 1 .. 10 ){
2       if (($login, $passwd, $uid)=getpwuid($num)){
3       print "$login--$uid\n";}
    }

(Output)
3   daemon--1
    sys--2
    bin--3
    uucp--4
    news--6
    ingres--7
    audit--9
```

解释

1. foreach 循环列表中含有一组范围从 1 到 10 的 uid 号，依次处理其中每一个号码。
2. getpwuid 函数从 /etc/passwd 检索信息，并将信息的前三个字段分别存储在数组元素 $login、$passwd 和 $uid 中。
3. 打印登录名及其相应的 uid。

18.2.7 时间和进程

当程序在计算机环境中工作时，经常需要获取并操纵当前的日期和时间。UNIX 系统维护了两种类型的时间值：日历时间（calendar time）和进程时间（process time）。

日历时间是从 UTC（Coordinated Universal Time，是 Greenwich Mean Time 的又一个名称）时刻 1970 年 1 月 1 日 00:00:00 开始计时而得到的秒数。

进程时间又称作 CPU 时间，它通过时钟时间、用户 CPU 时间和系统 CPU 时间来测量进程使用的资源情况。CPU 时间的计量单位是每秒的时钟滴答（tick）数。

Perl 提供了一些和时间有关的函数。它们能与系统交互，并检索其时间信息。

times 函数。times 函数返回一个 4 元素数组，其中含有进程的 CPU 时间信息，包括：

- 用户时间——用于执行用户代码的时间
- 系统时间——用于执行系统调用的时间
- 子进程的用户时间——所有已终止的子进程的总执行耗时
- 子进程的系统时间——所有已终止的子进程执行系统调用的耗时

格式

```
($user, $system, $cuser, $csystem) = times;
```

示例 18.46

```
(The Script)
    #!/bin/perl
1   printf "User time in this program %2.3f seconds\n", (times)[0];
2   printf "System time in this program %2.3f seconds\n", (times)[1];

(Output)
3   User time in this program 0.217 seconds
4   System time in this program 0.600 seconds
```

解释

1. times 函数返回一个 4 元素数组。打印其中第一个元素，即用户时间。
2. times 函数返回一个 4 元素数组。打印其中第二个元素，即系统时间。

time 函数（UNIX 和 Windows）。time 函数返回从 UTC 1970 年 1 月 1 日开始计时而得到的非闰秒数。读者可以借助 gmtime 函数和 localtime 函数使用其返回值，以便将时间设置为可供人阅读的格式。此外，当比较文件修改时间与访问时间时，stat 和 utime 函数也用到了 time 函数。

gmtime 函数。gmtime 函数会把 time 函数的返回值转换为由 UTC 字符构成的一个 9 元素数组。如果读者曾经是 C 程序员，就应当想到这些值是模仿了头文件 /usr/include/time.h 中的 tm 结构（详见表 18-11）。

格式

```
gmtime(EXPR);
gmtime EXPR;
($sec, $min, $hour, $monthday, $month, $year, $weekday,
            $yearday, $isdaylight)=gmtime;
```

表 18-11 gmtime 函数返回值

列表元素	涵 义
$sec	分之后的秒：[0, 59]
$min	小时之后的分：[0, 59]
$hour	从午夜开始计算的小时数：[0, 23]
$monthday	月份里的天数：[1, 31]
$month	从 1 月开始计算的月份：[0, 11]
$year	从 1900 年开始计算的年份
$weekday	从周日开始计算的天数：[0, 6]
yearday	从 1 月 1 日开始计算的天数：[0, 365]
isdaylight	夏令时标志

示例 18.47

```
(The Script)
    #!/bin/perl
```

```
1   ($sec, $min, $hour, $monthday, $month, $year, $weekday, $yearday,
         $isdaylight) = gmtime;
2   print "The weekday is $weekday and the month is $month.\n";
3   print "The time in California since midnight is ",
          `date "+%H:%M"`;
4   print "The Coordinated Univeral Time is $hour:$min
          since midnight\n";
5   print "Daylight saving is in effect.\n" if $isdaylight;
(Output)
2   The weekday is 2 and the month is 6.
3   The time in California since midnight is 20:35.
4   The Coordinated Univeral Time is 3:35 since midnight.
5   <no output>
```

解释

1. gmtime 函数返回表 18-11 中定义的数组。
2. 以 UTC 格式打印当前是星期几和当前的月份。
3. 利用 UNIX 的 date 命令打印加利福尼亚州时间。
4. 打印 UTC 时间。
5. 如果使用夏令时，则将 $isdaylight 设置为非 0 值。由于这里夏令时是无效的，所以不会打印任何内容。

localtime 函数。localtime 函数会把 UTC 时间转换为本地时区的一个 9 元素数组。

格式

```
localtime(EXPR);
localtime EXPR;
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
          $isdst)=localtime(time);
```

示例 18.48

```
(At the Command Line)
1   $ perl -e "print scalar (localtime);"
    Wed May 30 15:26:16 2007

(In Script)
2   $localtime=localtime;
    print $localtime;
```

解释

1. 这里的 localtime 函数用在标量上下文语境中，其返回值类似于 UNIX/Win32 date 命令的输出内容。scalar 函数强制使用标量上下文语境。
2. 将 localtime 的返回值赋值给标量。

示例 18.49

```
(The Script)
    #!/bin/perl
1   ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst)=
           localtime(time);
2      % weekday=(
       "0"=>"Sunday",
```

```
        "1"=>"Monday",
        "2"=>"Tuesday",
        "3"=>"Wednesday",
        "4"=>"Thursday",
        "5"=>"Friday",
        "6"=>"Saturday",
     );
      if ( $hour > 12 ){
3         print "The hour is ", $hour - 12 ," o'clock.\n";
     }
     else {
         print "The hour is $hour o'clock.\n";
     }
4  print qq/Today is $weekday{"$wday"}.\n/; # day starts at zero
5  print "It is ",$mon + 1, "/$mday/" , 1900+$year,".\n";
6  print "The isdst is $isdst.\n";

(Output)
3  The hour is 1 o'clock.
4  Today is Wednesday.
5  It is 5/30/2007.
6  The isdst is 1.
```

解释

1. localtime 函数把 time 函数的返回值转换为本地时间。
2. 关联数组 %weekday 把星期一到星期日分别关联到一周内的各天上面。
3. 打印小时和分。
4. 从 localtime 函数返回的标量 $wday 是一个数字，负责从 0 开始表示星期几。把该数字当作关联数组 %weekday 的键来使用，以便获得字符串 Wednesday 的值。
5. 打印年、月、日。
6. 如果夏令时有效，数组元素 $isdt 打印为 1；否则为 0。

18.2.8 UNIX 进程创建

Perl 程序在开始执行时会发生什么？这里简要介绍一下。一般而言，用户都是从 shell 命令行下执行 Perl 程序的。输入 Perl 脚本名（及其参数），然后按下回车键。这时 shell 就开始工作了。首先它会创建（fork）一个叫作子进程的进程。这个子进程实质上就是创建它的 shell 的副本。现在就有两个进程正在运行，即父进程和子进程。当创建好子进程后，shell 父进程一般就会进入休眠状态（waits），并由子进程负责准备执行 Perl 程序；即，处理管道重定向（如果需要的话）、执行后台处理工作，等等。当 shell 子进程完成了准备工作后，就会在其内部开始执行（execs）Perl 程序。当 Perl 进程执行完毕时，它也会退出（exits）执行，并将其退出状态返回给正在等待的 shell 父进程。这时，shell 会再次醒来，并在屏幕上出现命令行提示符。如果用户输入的是 UNIX 命令，也将重复上述流程。

可以想象，有的 Perl 程序也需要启动子进程，以便处理特定的任务，譬如数据库应用程序或客户端 / 服务器程序。

fork 函数。 fork 函数负责在 UNIX 系统上创建进程。调用 fork 进程一次会返回两次结果。它创建父进程（调用进程）的副本。新的进程又称作子进程。子进程继承了父进程的上下文、打开的文件、真实用户 ID、掩码、当前工作目录、信号等。父子进程都执行相同的代码，从 fork 函数调

用之后，指令才开始执行。

fork 函数可以区分父进程和子进程，因为它会对不同的进程返回不同的值，对于子进程它会返回 0，而对父进程则返回其子进程的 pid。调用 fork 函数后，它不能保证首先执行其中的某个进程。

wait、exec 和 exit 函数一般都和 fork 函数配合使用，因此可以同时控制父进程和子进程。例如，让父进程等待子进程完成某项任务，并在子进程退出后恢复该父进程。

图 18-6 说明了 UNIX shell 是如何使用 fork 系统调用来创建新进程的。当用户在 shell 提示符下输入 Perl 程序名后，shell 会孵化一个进程，并创建其本身的副本，又称作子进程。然后父进程进入休眠（wait）状态。子进程代替它执行 Perl 进程。子进程始终不返回。请注意，ENV 环境变量、标准输入、标准输出和标准错误消息都是继承得来的。Perl 程序完成后就退出，父 shell 恢复执行。最终，shell 提示符重新出现在屏幕上。Perl 程序也可使用 fork 函数孵化出其他的应用程序。

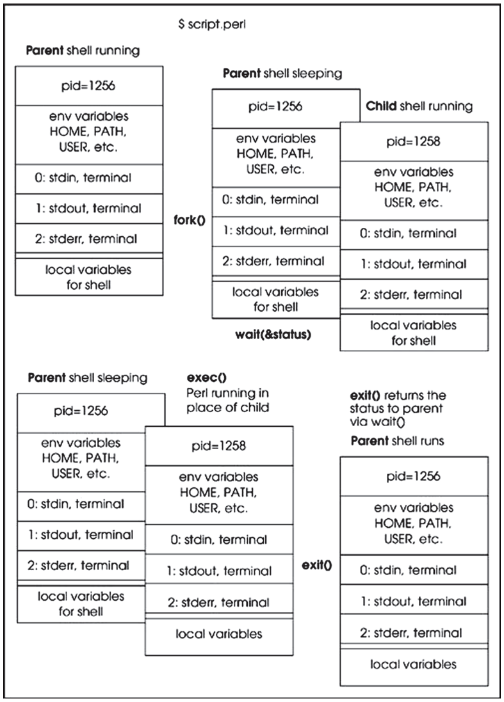


图 18-6　从 shell 创建 Perl 进程

格式

```
fork;
```

示例 18.50

```
(The Script)
$return_val=fork;
if ( $return_val == 0 ){
   print "This is the child process; return value
         is $return_val.\n";
}
elsif ( defined $return_val ){
```

```
            print "This is the parent process; return value
                  is $return_val.\n";
        }
        else{
4           die "fork error: $!\n";
        }

(Output)
2   This is the child process; return value is 0.
3   This is the parent process; return value is 3512.
```

解释

1. 调用 fork 函数创建该进程的副本。
2. 检查返回值。如果返回值为 0，则执行子进程代码。
3. 如果返回值非 0，则执行父进程。
4. 如果 fork 函数调用失败，则执行本行语句。如果进程表已满，该语句也可能会失败；即，系统达到了所允许的最大进程数。

exec 函数。fork 函数能创建新的进程，而 exec 函数则能在当前进程中启动新的程序。一般都是在 fork 函数之后调用 exec 函数的。Perl 是从 shell 中继承属性的，而从 Perl 执行的进程也将继承 Perl 的属性，如 pid、gid、uid、信号、目录等等。如果在 Perl 脚本中直接调用 exec 函数，则新程序会代替当前正在运行的 Perl 程序执行。当程序运行完毕后，不返回 Perl 程序。由于 exec 函数不会清除输出缓冲区，因此必须手动设置变量 $1，以确保顺利地缓冲命令。

在调用 exec 函数后，文件句柄 STDIN、STDOUT 和 STDERR 仍将保持打开状态。

在系统层面，有六种不同的 exec 函数可以用于初始化新程序。如果给 exec 函数传递了多个参数，Perl 会调用 C 的 execvp 库函数。其参数是执行程序名和传递给该程序的其他参数。如果给 exec 函数传递单个参数，并且其中含有任意的 shell 元字符，Perl 则会将它传送给 shell 命令 /bin/sh -c 来解释。

格式

```
exec(UNIX COMMAND);
exec UNIX COMMAND;
```

示例 18.51

```
(The Script)
1   exec 'echo hi there you!';
2   print "hello";

(Output)
1   hi there you

(The Script)
1   exec 'ls *.c';
2   print "hello.";

(Output)
file1.c file2.c file3.c
```

解释

1. 在这两个示例中，exec 函数都执行了 UNIX 命令。
2. 在这两个示例中，print 语句都没有执行，因为 exec 始终不返回。本例执行了 UNIX 命令，而不是 Perl 程序。

wait 和 waitpid 函数。wait 函数负责等待子进程执行完毕。在调用 fork 函数后，父进程和子进程都将得以执行，wait 函数将强制父进程等待，直到子进程执行完毕，并将子进程的 pid 返回给父进程。如果子进程不存在，wait 就返回 −1 ⊖。

格式

```
wait;
```

示例 18.52

```
(The Script)
1   $return_val=fork;
2   if ( $return_val == 0 ){         # In child
       print "This is the child process; return value
           is $return_val.\n";
3      exec "/bin/date" || die "exec failed: $!\n";
    }
4   elsif ( defined $return_val ){    # In parent
       print "This is the parent process; return value is $pid.\n";
5      $pid = wait;
       print "Back in parent process.\n";
       print "The deceased child's pid is $pid.\n";
    }
    else{
6      die "fork error: $!\n";
    }

(Output)
4   This is the parent process; return value is 3530.
2   This is the child process; return value is 0.
3   Wed May 30 23:57:18 PST 2007.
5   Back in the parent process.
    The deceased child's pid is 3530.
```

解释

1. fork 函数创建当前进程的副本。现在就有两个进程同时运行：父 Perl 进程和子 Perl 进程。二者都在调用 fork 后直接开始执行代码。在子进程中返回值是 0，而在父进程中的返回值则是子进程的 pid。
2. 如果是在子进程中，执行 if 语句块。
3. exec 函数执行 UNIX 的 date 命令，并且不返回。
4. 如果是在父进程中，执行 elsif 语句块。$return_val 的值是子进程 pid。
5. 父进程调用 wait 函数等待子进程执行完毕。返回完成子进程的 pid。
6. 如果 fork 失败（无法创建更多进程？），die 函数负责打印错误信息并退出程序。

⊖ waitpid 函数也能等待子进程完成，不过它能指定等待哪一个子进程，它还拥有控制阻塞的特殊标志位。

exit 函数。exit 函数负责退出程序。读者可以为它提供范围从 0 到 255 的整数型参数。exit 值将通过 wait 函数返回给父进程。一般而言，以状态 0 退出的 UNIX 程序就是成功的，而以非 0 状态退出的程序则是失败的程序（当然，程序成功与否的标准并不是相同的）。

格式

```
exit (Integer);
exit Integer;
```

示例 18.53

```
(The Script)
    # The name of the script is args.p
1   exit 12 if $#ARGV == 0;

(Output)
1   $ args.p
2   $ echo $?
    12
```

解释

1. 脚本缺少参数。
2. shell 的 $? 变量含有 Perl 程序的退出状态。如果使用 C shell，则退出状态会保存在 $status 中。

18.2.9 Win32 进程创建

在 Win32 系统上，读者也可以像在 UNIX 系统上一样使用 system 函数、exec 函数和反引号。

start 命令。Windows 和 UNIX 都使用 system 函数启动操作系统命令。system 负责执行程序，并在程序完成之前不返回。如果将 Windows start 命令作为参数提供给 Perl 的 system 函数，便会运行新的应用程序，同时脚本仍将继续执行。

示例 18.54

```
    use warnings;
1   $return_value = system ("start /Program
Files/Netscape/Communicator/Program/netscape.exe");

2   print "Program continues; Netscape is running.\n";
3   print "The return_value from system is $return_value.\n";

(Output)
Program continues; Netscape is running.
The return_value from system is 0.
```

解释

1. Perl 的 system 函数启动新的进程。通过使用 Win32 start 命令，可以启动新的进程。与此同时，Perl 脚本还将继续运行，而不是等待新进程执行完毕。如果进程启动成功，system 函数将返回 0。
2. Netscape Communicator 已经启动，并将继续运行，直到关闭所有窗口。
3. 打印 system 函数的返回值。

Win32::Spawn 模块。 Win32::Spawn 函数的行为类似于 system 函数和 Windows start 菜单。

格式

```
use Win32;
Win32::Spawn($ProgramName, $CommandLine,$ProcessID);
```

示例 18.55

```
1   use warnings;
2   use Win32;
3   $|=1;
4   $Application="C:/mksnt/date.exe";
5   $CommandLine="date +%D";
6   $status=Win32::Spawn($Application, $CommandLine, $ProcessID);
7   if ($status != 0){
8      print "pid is $ProcessID.\n";
    }
    else{
9      print "Didn't spawn $Application.\n";
10     print Win32::FormatMessage(Win32::GetLastError);
    }

(Output)
8 pid is 448.
11/01/00
```

解释

1. warnings 编译指示符将负责发送语法警告、未初始化的值等等，并对程序中可能发生的问题提供帮助信息。
2. 加载 Win32 模块。这是 ActiveState 带有的模块，读者可在 site/lib/Win32 目录下的标准 Perl 库中找到它。它含有许多对 Windows 程序员而言非常有用的模块，能够处理服务器、客户端、注册表、时间、网络管理、进程等任务。
3. 将特殊标量 $1 赋值为 1，以确保在使用了 print 函数之后及时清空输出缓冲区。
4. 应用程序是位于 C:\drive 下的 MSK 工具包[a]。该工具包含有模拟 UNIX 命令的 .exe 文件。date.exe 文件能得到当前的日期。
5. 在命令行中含有传递给应用程序的参数。
6. 以三个参数调用 Win32::Spawn 函数。其中两个参数已经给出了值，而 $ProcessID 的值仍然未知，它将在应用程序启动后给出。
7. 如果 Win32::Spawn 函数返回的状态值为 0，则打印进程 ID 号。
8. 进程 ID 号是 448。
9. 如果 .exe 文件不退出，则显示 Didn't spawn C:/mksnt/date.exe。
10. 如果因为应用程序没有退出而不能启动进程，则 GetLastError 函数会打印：

```
The system cannot find the file specified.
Program continues to run....
```

这是 Windows 生成的错误信息，是由操作系统无法启动程序而引起的。FormatMessage 函数能为错误信息创建一份可读的输出。

a. Windows NT 和 Windows 95 上的 MSK 工具包，6.1 版，Mortice Kern Systems Inc.

Win32::Process 模块。 用于运行 Windows 应用程序的另一种扩展模块则是面向对象的 Win32::Process 模块（详见表 18-12）。该模块提供了很多可用于创建和管理进程的方法。用户可使用该模块实现进程的挂起、恢复和撤销。

表 18-12 Win32::Process 方法

方 法	用 途
Create($Obj, $AppName, $CommandLine, $Inherit, $CreateFlags, $InitialDir);	创建进程对象
$Obj->GetExitCode($ExitCode);	获取进程退出代码
$Obj->GetPriorityClass($Class);	获得进程优先级类
$ProcessObj->GetProcessID();	获得进程 ID
$Obj->Kill($ExitCode);	使用退出代码撤销进程
$Obj->Resume();	恢复挂起的进程
$Obj->SetPriorityClass($Class);	设置进程优先级掩码（NT）
$Obj->Suspend();	挂起进程
$Obj->wait($Timeout);	等待进程执行完毕

示例 18.56

```
1   use Win32::Process;
2   use Win32;
3   sub ErrorReport{
        print Win32::FormatMessage( Win32::GetLastError() );
    }

4   Win32::Process::Create($ProcessObj,
        "C:\\windows\\notepad.exe", "notepad myfile.txt", 0,
        NORMAL_PRIORITY_CLASS, ".") || die ErrorReport();
    print "Notepad has started\n";
5   print "The exit code is:
        ",$ProcessObj->GetExitCode($ExitCode),"\n";

(Output)
Notepad has started
The exit code is: 1
```

解释

1. 加载 Win32::Process 模块，该模块用户运行 Windows 应用程序。
2. 加载 Win32 模块。
3. ErrorReport 函数负责发送上一次发生的系统错误的格式化错误信息。
4. Create 函数创建进程。第一个参数 $ProcessObj 是进程对象容器，其后的参数包括应用程序的全路径、命令行参数和必需标志位。
5. 返回进程的退出代码。

18.3 与操作系统交互的其他途径

如果系统函数仍不能满足需求的话，Perl 还提供了其他可供选择的操作系统处理途径。读者还

可使用 syscall 函数、命令替换、system 函数和 here 文档来获得系统信息。

18.3.1 syscall 函数和 h2ph 脚本

syscall 函数会根据其参数调用指定的系统调用。如果系统无法执行 C 系统调用，则该函数会返回致命错误。其第一个参数是系统调用的名字，以 &SYS_ 为前缀。其余参数则是真实系统调用所要求的实际参数。如果参数是数值型的，则通过 C 整数形式予以传递。如果不是，则传递指向字符串值的指针。如果参数不是实际的值，且无法在上下文语境中予以解释，则必须通过给整数添加 0 的方式把数组强制变换为整数。

在使用 syscall 函数之前，首先应当运行 Perl 发布包自带的 h2ph 脚本（在 Windows 上是 h2ph.bat）。h2ph 脚本的底部（即 _END_ 之后）提供了 h2ph 的手册页，说明了如何运行该脚本。该脚本负责把适当的 C 头文件转换为对应的 Perl 头文件。如果所用函数要求使用它们，则必须将这些文件添加到 Perl 库中。创建的所有文件都应当拥有 .ph 扩展名。在运行 h2ph 脚本后，还应确保程序中的 @INC 数组含有指向这些库函数的路径[⊖]。

格式

```
syscall (&SYS_NAME, LIST);
```

示例 18.57

```
(UNIX: At the Command Line)
1   $ cd /usr/include; /usr/local/bin/perl/h2ph * sys/*

(In Script)
    #!/bin/perl
    # The name of the script is args.p
2   push(@INC, "/usr/local/lib");
3   require "syscall.ph";
4   $bytes=syscall(&SYS_getpagesize);
5   printf "The pagesize for this Sparc Sun Workstation is %d
           bytes \n",$bytes;

(Output)
5   The pagesize for this Sparc Sun Workstation is 4096 bytes.
```

解释

1. 执行 h2ph 脚本，将所需 C 头文件转换为相应的 Perl 头文件。创建的文件将放到 /usr/local/lib 中，并以 .ph 作为扩展名。
2. 将含有 .ph 文件的目录载入 @INC 数组中。
3. 使用 C 系统调用，需要用到 syscall.ph 文件。
4. Perl syscall 函数会调用 getpagesize 系统调用。前缀 &SYS_ 是必须出现的 Perl 语法，必须预先附加到真实的系统调用名上。
5. Sun4c 的分页大小为 4096 字节。

⊖ 在 Perl 5 发布包中也可看到 h2xs 脚本，用于从 C 头文件中建立 Perl 扩展。

18.3.2 命令替换——反引号

在前面第 4 章中，本书已经讨论了命令替换和反引号。由于命令替换是与操作系统交互的另一种方法，因此这里再简要地作一下回顾。

UNIX shell 使用反引号来执行命令替换，在 Perl 脚本中的执行方式也与之类似。例如，命令行 echo The present working directory is 'pwd' 会执行反引号中的命令，并将其结果替换为字符串。与 UNIX shell 相同，把命令放在反引号之间就会导致它的执行。与 shell 不同的是，如果用双引号包围反引号字符串，命令替换动作就不会发生。执行命令所生成的输出内容将保存到标量变量中。

示例 18.58

```
(The Script)
    #!/bin/perl
1   print "The hour is ",'date';

2   @d='date';
3   print $d[0]'

4   @d=split(/ /,'date');
5   print "$d[0]\n"
6   $machine='uname -n';
7   print "$machine\n"'

(Output)
1   The hour is Thu May 31 20:47:17 PDT 2007
3   Thu May 31 20:59:11 PDT 2007
5   Thu
7   dolphin
```

解释

1. UNIX 的 date 命令位于反引号之间。执行该命令，并将它附加到字符串 The hour is 后面。
2. 将数组 @d 赋值为 date 命令的输出内容。该属主以单个字符串的形式存储，与 shell 不同。后者的输出内容是保存为列表的。
3. 由于命令的输出内容是以单个字符串形式存储的，因此 $d[0] 就是整个字符串。
4. split 命令根据 date 命令返回的输出内容创建列表。
5. 这时，数组的第一个元素也就是列表的第一个字，即 Thu。
6. 将标量 $machine 赋值为 UNIX 命令 uname -n，其中含有主机名（在 BSD 上可使用 hostname 命令）。
7. 打印主机名。

18.3.3 Shell.pm 模块

通过该模块，用户可以在 Perl 脚本中使用类似于在 shell 提示符下输入的那些 UNIX 命令。这些命令都将当成 Perl 子例程来对待。其参数和选项将以字符串列表的形式传送给相应命令。

示例 18.59

```
(The Script)
    #!/bin/perl
1   use Shell qw(pwd ls date);   # Shell commands listed
```

```
2   print "Today is ", date();
3   print "The time is ", date("+%T");
4   print "The present working directory is ", pwd;
5   $list=ls( "-aF");
6   print $list;

(Output)
2   Today is Tue May 29 13:41:56 PDT 2007
3   The time is 13:41:57
4   The present working is /home/ellie/sockets
6   ./
    ../
    sh.test*
    shellstuff
    timeclient*
    timeclient5*
    timeserver*
    timeserver5*
```

解释

1. 在该程序中使用 Shell.pm 模块。pwd、ls 和 date 这三个 UNIX 命令将当作普通的 Perl 子例程来对待。
2. 将 date 命令作为子例程执行。这是对反引号机制的另一种实现。
3. 传递给 date 命令的参数是位于引号中的字符串。
4. 执行 pwd 命令。
5. 将 ls 命令的输出内容赋值给标量 $list。其参数是以单个字符串的形式传递给函数的。
6. ls -aF 命令会输出当前工作目录中的文件列表。-a 开关项表明包含点文件，而 F 则表明为可执行文件标记星号（*），并对目录标上 / 号。

18.3.4 system 函数

与 C 中的相应函数类似，system 函数也是以系统命令作为其参数点。它将把命令发到系统 shell 上解释，并将控制权返回给调用程序，即脚本。该函数的功能与 exec 几乎相同，所不同的是它首先会调用 fork 函数，因此其控制权会返回给 Perl 脚本。由于它不会自动清空输出缓冲区，因此需要将特殊 Perl 变量 $| 设置为 1，以便在 print 和 write 语句之后强制清空缓冲区[⊖]。

格式

```
system("system command");
system "system command";
```

示例 18.60

```
(UNIX: The Command Line)
1   system("cal 1 2007");
    print "Happy New Year!\n";'

(Output)
    January 2007
```

⊖ fork 执行完毕后，脚本将等待命令执行，并将控制权返回给脚本。

```
Su  Mo  Tu  We  Th  Fr  Sa
 1   2   3   4   5   6
 7   8   9  10  11  12  13
14  15  16  17  18  19  20
21  22  23  24  25  26  27
28  29  30  31
    Happy New Year!

(Windows)
2   system("notepad.exe");
```

解释

1. system 函数执行 UNIX 的 cal 命令，打印出 2007 年 1 月的日历信息。
2. system 函数执行 Windows 的 notebook.exe 命令，并启动 Notepad 记事本会话。

示例 18.61

```
     (The Script)
     1   print "Hello there\n";
     2   print "The name of this machine is ";
     3   system ("uname -n"); # Buffer is not flushed
     4   print "The time is ", 'date';

     (Output)
1    Hello there
3    jody
2,4  The name of this machine is The time is Tue May 29 13:39:35 PDT 2007
```

解释

1. 按计划执行第一个 print 语句。
2. 由于 Perl 依赖于默认的 I/O 缓冲机制，因此在执行完 print 后可能并没有立刻清空缓冲区；首先打印由 shell 负责执行的 system 函数的调用结果。
3. system 函数令 shell 执行 UNIX 命令 uname -n。
4. 直接在第 2 行的 print 语句之后打印 print 语句。

示例 18.62

```
(The Script)
    #!/bin/perl
1   $|=1; # Set special variable to flush the output buffer
2   print "Hello there\n";
3   print "The name of this machine is ";
    system ("uname -n");
4   print "The time is ", 'date';

(Output)
2   Hello there
3   The name of this machine is jody
4   The time is Tue Jan 26 13:43:54 PST 2001
```

解释

当特殊 Perl 变量 $| 的值不是 0 时，将在每个 print 与 write 语句之后强制清空缓冲区。

18.3.5 here 文档

Perl 的 here 文档机制是从 UNIX shell 中的 here 文档机制派生而来的。和在 shell 中一样，Perl 中的 here 文档也是面向行的引用表单，要求提供 << 运算符，其后跟随一个初始的终止字符串。<< 之后可以不出现空格。如果终止字符串没有加上引号或双引号，则允许执行变量表达式。如果终止字符串上加了单引号，则不执行变量表达式。用户应当把文本的第一行内容插入到第一个和最后一个终止字符串之间。最后一个终止字符串必须位于该行上，且周围不允许出现空白字符。

与 UNIX shell 不同的是，Perl 不允许在 here 文档中执行命令替换。另一方面，如果将终止字符串包含在反引号中的话，Perl 也允许在 here 文档中执行该命令。

示例 18.63

```
(The Script)
    #!/bin/perl
    $price=100;
1   print <<EOF;    # No quotes around terminator EOF are same
                    # as double quotes
2   The price of $price is right.     # Variables are expanded
3   EOF
4   print <<'FINIS';
5   The price of $price is right.
                    # The variable is not expanded
                    # if terminator is enclosed in single quotes
6   FINIS

7   print << x 4;   # Prints the line 4 times
8   Christmas is coming!
        # Blank line is necessary here as terminating string

9   print <<'END';  # If terminator is in backquotes,
                    # will execute UNIX commands
10  echo hi there
11  echo -n "The time is "
12  date
13  END

(Output)
2   The price of 100 is right.
5   The price of $price is right.
8   Christmas is coming!
    Christmas is coming!
    Christmas is coming!
    Christmas is coming!
10  hi there
    The time is Fri Nov 3 17:03:46 PST 2000
```

18.3.6 Globbing（文件名扩展和通配符）

在使用 UNIX 和 MS-DOS 命令行时，已经引入了一些 shell 元字符，它们用于扩展文件名。其中，星号（*）用于匹配文件名中的所有字符，问号（?）负责匹配文件中的单个字符，而方括号则匹配文件名中某一组字符中的一个。将这些 shell 元字符扩展为文件名的过程就称作 globbing。

如果将文件名放到读运算符（角括号）中，则 Perl 会自动支持 globbing。Perl 5 中也提供了一个专用于 globbing 的函数，如示例 18.64 所示。

示例 18.64

```
(The Script)
    #!/bin/perl
1   @myfiles=<*.[1-5]>;
2   print "@myfiles\n";
3   foreach $file ( <p??l[1-5]*>){
4      print "$file\n" if -T $file;
    }
(Output)
2   exer.3 exer.4 exer.5 fileter.1 format.1 format.2 format.3 perl.4
perl.4.1
4   perl1
    perl2
    perl3
    perl4
    perl4.1
    perl5
```

解释

1. 在数组上下文语境中，当执行完 globbing 之后，返回含有所有匹配文件的列表，并返回数组 @myfiles。该列表由文件构成，其中每个文件名都以 0 到多个字符开头，其后是一个点号，末尾是 1 到 5 之间的数字。
2. 打印匹配文件的列表。
3. 进入 foreach 循环。每次循环时，就将 $file 设置为成功执行 glob 操作的下一个文件；即符合下列条件的任意文件：其文件名以 p 开头，后面跟随任意两个字符，再后是一个 l，再后则是 1 到 5 之间的数字，并以零到多个任意字符结尾。
4. 如果该文件是一个文本文件（-T），打印其文件名。

glob 函数。glob 函数与运算符 <*> 的作用完全相同。它能扩展文件名中的元字符，并返回扩展后得到的文件名。

示例 18.65

```
(Command Line)
1   $ perl -e 'while(glob("p???[1-5]")) {print "$_\n";}'
    perl1
    perl2
    perl3
    perl4
    perl5

(In Script)
2   while ( glob("p???[1-5]")){
3       print "$_\n";
    }
(Output)
3   perl1
    perl2
```

```
perl3
perl4
perl5
```

解释

1. 在命令行中，glob 函数能“glob”当前目录中符合条件的任意文件，这些文件的名字应当以 p 开头，随后是三个任意字符（即 ???），最后是 1 到 5 之间的数字（即 [1-5]）。将匹配这个表达式的所有文件名赋值给 $_，并将它们打印出来。
2. 这一次在脚本中使用 glob，其行为与第一个示例完全一样。
3. 打印扩展得到的文件名。

18.4 错误处理

在很多情况下，系统调用可能会失败；例如，尝试打开不存在的文件，或者删除某个仍含有文件的目录，或者尝试读取没有读权限的文件。在前面的示例中，我们已经用到了 die 函数，本节将详细讨论有关错误处理和错误处理函数的相关内容。这些函数包括 die 函数、warn 函数和 eval 函数。

- die 函数用于在命令或文件句柄失败时退出 Perl 脚本。
- warn 函数类似于 die 函数，但它不会退出脚本。
- eval 函数具有多种用途，但它主要还是用于异常处理。

读者想必还记得短路运算符 && 和 ||，这两个运算符首先会求其左侧操作数的值，然后才会求其右侧操作数的值。如果 && 左侧操作数值为 true，则求其右侧的操作数。如果 || 左侧操作数的值为 false，这才求其右侧的操作数。

Carp.pm 模块。有很多种退出脚本的途径可供用户选择。Perl 5 提供的 Carp 模块扩展了 die 和 warn 的功能。（详见示例 12.10。）

18.4.1 die 函数

如果系统调用失败的话，die 函数会把字符串打印到 STDERR 上，并以 $! 的当前值退出脚本。$! 变量中含有 errno 的当前值，后者是一个 UNIX 全局变量，含有一个表示系统错误的数字。只有在系统调用失败时才会更新 errno 的值。当系统调用失败时，会向 errno 赋予一个数字代码，以表明错误的类型。如果在字符串中省略了换行符，则会打印出带有行号的消息（参见 /usr/include/sys 中的完整列表）。

下面是 /usr/include/sys/errno.h 文件中的示例：

```
#define EPERM      1   /* Not owner */
#define ENOENT     2   /* No such file or directory */
#define ESRCH      3   /* No such process */
#define EINTR      4   /* Interrupted system call */
#define EIO        5   /* I/O error */
...
```

Win32 的错误代码不同于 UNIX 错误代码，因此不能依赖于 $! 返回的值。有很多 Win32 扩展都提供了自己的错误函数，以便为用户提供更有意义的结果。详情可参阅 ActiveState 中标准 Perl 库里的 Win32::GetLastError 相关文档。

格式

```
die(LIST)
die LIST
die
```

示例 18.66

```
(In Script)
1   die "Can't cd to junk: $!\n" unless chdir "/usr/bin/junk";

(Output)
1   Can't cd to junk: No such file or directory
```

解释

chdir 调用失败。$! 中含有来自 errno 的错误消息。换行符导致打印了 die 函数后面的字符串，该字符串中含有变量 $! 的值。

示例 18.67

```
(In Script)
1   die unless chdir '/plop' ;

(Output)
1   Died at croak.perl line 4.
```

解释

chdir 调用失败。这一次 $! 并不在 die 字符串中，打印发生错误的行。

示例 18.68

```
(In Script)
1   chdir '/plop' or die "Stopped";

(Output)
1   Stopped at croak.perl line 4.
```

解释

本示例的输出内容与前一个示例相同，但它使用了不同的语法。如果 chdir 调用失败，就执行 or 右边的 die 函数。

18.4.2 warn 函数

warn 函数（运算符）和 die 几乎一样，所不同的是前者会让程序继续运行。如果在 eval 块中调用 die 函数，则传递给 die 的参数字符串将同样赋值给特殊变量 $@。在调用 die 之后，该变量可以作为一个参数传递给 warn 函数，其输出将发送到 STDERR 上（参阅“ecal 函数”一节）。

18.4.3 eval 函数

eval 函数用于处理异常，即捕捉错误。位于 eval 之后的语句块是作为单独的 Perl 程序来处理和解析的，但其所有的变量设置、子例程以及格式定义都将保持到 eval 执行完毕。

eval 函数返回的值是上一个表达式的值。如果出现了编译或运行时错误，或者执行了 die 语句

的话，则会返回未定义的值，并将特殊变量 $@ 设置为错误消息内容。如果没有发生错误，则 $@ 将是空字符串。

借助 eval 求 Perl 表达式的值

示例 18.69

```
(The Script)
    #!/bin/perl
    # The eval function will evaluate each line you type
    # and return the result. It's as though you are
    # running a little independent Perl script.
    # Script name: plsh

1   print "> ";         # Print the prompt
2   while(<STDIN>){
3       $result=eval ;  # eval evaluates the expression $_
4       warn $@ if $@;  # If an error occurs, it will be assigned to $@
5       print "$result\n if $result";
6       print "> "; # Print the prompt
    }
(Output)
(The Command line)
    $ plsh
2   > hello
5   hello
2   > bye
5   bye
2   > 5 + 4
5   9
2   > 8 / 3
5   2.66666666666667
2   > 5 / 0
4   Illegal division by zero at (eval 5) line 3, <STDIN> line 5.
    > "Oh I see
    Can't find string terminator '"' anywhere before EOF at (eval 6)
       line 1,  <STDIN> line
    > exit
```

解释

1. 本行向用户打印出一个提示信息。该程序类似于小型 Perl shell。它能帮助用户在把表达式放进程序之前检查它的好坏，特别是在不确定 Perl 会如何处理该表达式时。
2. 进入 while 循环。每次进入循环时，就从用户处读取一行输入，并赋值给 $_。
3. 不带参数的 eval 将对 $_ 中的表达式求值，并将求值结果赋予 $result。
4. 如果 eval 发现了求值表达式引起的语法错误或系统错误的话，就将返回的错误信息赋值给变量 $@。如果没有发现错误，则把 $@ 赋值为空字符串。
5. 如果对表达式求值成功，打印其结果。
6. 显示提示信息，并再一次进入循环。

使用 eval 捕捉程序中的错误

示例 18.70

```
(In Script)
    #!/bin/perl
    print "Give me a number.";
```

```
    chop($a=<STDIN>);
    print "Give me a divisor.";
    chop($b=<STDIN>);
1   eval{ die unless $answer = $a/$b ; };
2   warn $@ if $@;
3   printf "Division of %.2f by %.2f is %.2f.\n",$a,$b,
           $answer if $answer ;
4   print "I'm here now. Good-day!\n";

(Output)
    Give me a number.45
    Give me a divisor.6
3   Division of 45.00 by 6.00 is 7.50.
4   I'm here now. Good-day!

(Output)
    Give me a number.5
    Give me a divisor.0
2   Illegal division by zero at ./eval.p line 8, <STDIN> line 2.
4   I'm here now. Good-day!
```

解释

1. eval 函数会计算除法 ($a/$b)，并将结果保存到 $answer 中。请注意，必须首先在 eval 中使用 $answer，它会一直保留到 eval 执行结束。
2. 如果一切正常，并且除法运算顺利完成，则忽略这一行。如果发现错误（例如除以 0），则将 $@ 变量设置为系统错误信息，然后通过 warn 函数将消息打印到 STDERR，并恢复程序执行。如果在 eval 块中调用了 die 函数，则程序不退出，而是在退出 eval 块之后继续执行。
3. 如果运行成功，则打印除法运算的结果。
4. 打印该行内容只是为了说明，程序即使失败了也会继续执行，因为 warn 函数不会导致脚本退出。

eval 函数和 here 文档

示例 18.71

```
(The Script)
    #!/bin/perl
1   eval<<"EOF";
2       chdir "joker" || die "Can't cd: $!\n";
3   EOF
4   print "The error message from die: $@";
5   print "Program $0 still in progress.\n";

(Output)
4   The error message from die: Can't cd: no such file or directory
5   Program ./eval4.p still in progress.
```

解释

1. here 文档类似于一种特殊形式的引用。eval 函数获得位于第一个 EOF 和最后一个 EOF 之间的所有内容。
2. 如果 chdir 函数调用失败，则调用 die 函数，并在 here 文档的最后一个 EOF 之后恢复程序。
3. EOF 表明 here 文档到此结束。

4. 将 die 函数的错误消息保存在变量 $@ 中。
5. 程序继续执行。

18.5 信号

信号负责把消息发送给进程，一般会造成进程结束。它们通常是由某些意外事件触发的，譬如非法除 0、分段冲突（segmentation violation）、总线错误（bus error）或者电源失效等等。内核还把信号当作计时器来使用，例如向进程发送警告信号。当用户敲击 BREAK、DELETE。QUIT 或 STOP 键时，也会发出相应的信号。

内核能够辨别 31 种不同的信号，它们都列举在 /usr/include/signal.h 文件中。如需获取一份信号列表，读者可以直接在 UNIX 提示符下键入 kill -l 命令（见表 18-13）。

表 18-13　信号（BSD）*

名称	数字	默认值	描述
SIGHUP	1	Terminate	挂起
SIGINT	2	Interrupt	中断
SIGQUIT	3	Terminate	退出 / 生成核心文件
SIGILL	4	Terminate	终止

* 这只是部分信号的清单

捕获信号。信号是一种异步事件；也就是说，进程不知道信号将何时到达。用户可以通过编程方式去忽略某些进入进程的信号，或者设置信号处理程序在信号到达时执行特定的子例程。Perl 程序在关联数组 %SIG 中设置了需要处理的信号。如需要忽略某个信号，则应在调用 fork 或 exec 函数之后再忽略它。

用户还可对程序内的某个分段实现忽略或处理某个信号，然后再重置其默认行为。详见示例 18.72。

格式

```
$SIG{'signal'};
```

示例 18.72

```
(The Script)
    #!/bin/perl
    sub handler{
        local($sig) = @_;      # First argument is signal name
        print "Caught SIG$sig--shutting down\n";
        exit(1);
    }
    $SIG{'INT'} = 'handler'; # Catch <Ctrl>-c
    $SIG{'HUP'}='IGNORE';
    print "Here I am!\n";
    sleep(10);
    $SIG{'INT'}='DEFAULT';[a]
```

```
(Output)
7   Here I am
        < <Ctrl>-c is pressed while the process sleeps >
3   Caught SIGINT--shutting down
```

解释

1. 定义子例程 handler。
2. local 函数将其第一个参数（即信号名）设置为标量 $sig。
3. 如果信号到达，则执行处理程序例程，并打印该语句。
4. 程序以值 1 退出，表明出现了异常。
5. 设置 $SIG 关联数组的值。它的键是没有 SIG 前缀的信号名。它的值是需要调用的子例程名。如果在程序运行期间按下了 <Ctrl>-c 中断键，则会调用相应的处理例程。
6. IGNORE 表示忽略挂起信号。
7. 执行 print 语句，程序休眠 10 秒钟。如果信号 <Ctrl>-c 到达，就调用处理程序例程。
8. 将 SIGINT 信号重置为默认状态，以便在按下 <Ctrl>-c 时能终止程序。

向进程发送信号——kill 函数。 读者如需把信号发送到进程或进程列表的话，可使用 kill 函数。列表的第一个元素是数字值，如果加引号的话，则是信号名。该函数将返回成功接收该信号的进程号。如果信号数字是一个负数，就表明取消该进程组。如要取消进程，首先必须拥有该进程；也就是说，kill 信号发送进程和 kill 信号接收进程的有效 uid 与真实 uid 必须是相同的。

如需了解更为复杂的信号处理机制，请参阅 Perl 标准库中提供的 POSIX 模块。

格式

```
kill(LIST);
kill LIST;
```

示例 18.73

```
1   $ sleep 100&
2   $ jobs -l
    [1] + 6505 Running sleep 100&
3   $ perl -e 'kill 9, 6505'
    [1] Killed sleep 100
```

解释

1. 在 UNIX shell 提示符下，以后台方式执行 sleep 命令。sleep 命令将使进程暂停 100 秒。
2. jobs 命令会列出后台运行的进程。sleep 进程的 pid 为 6505。
3. 在命令行下执行 Perl。kill 函数的参数有两个。第一个是信号数值 9，以确保终止进程。第二个参数是 sleep 进程的 pid 号。

alarm 函数。alarm 函数负责让内核在规定秒数之后将 SIGALARM 信号发送到调用进程。如果在 alarm 信号仍然有效时调用 alarm 函数，则会覆盖前面已有的值。

格式

```
alarm (SECONDS);
alarm SECONDS;
```

示例 18.74

```
(The Script)
1   alarm(1);
2   print "In a Forever Loop!";
3   for (; ;){ printf "Counting...%d\n", $x++;}

(Output)
2   In a Forever Loop!
3   Counting...0
    Counting...1
    Counting...2
    Counting...3
    Counting...4
        ...
    Counting...294
    Counting...295
4   Alarm Clock
```

解释

1. 一秒钟后，将 SIGALARM 信号发送到该进程。
2. 打印该语句。
3. 开始循环。首先等待 1 秒钟。这里可以关闭对实际秒数的解析精度。读者可使用 syscall 函数调用其他的函数。例如，setitimer（2）和 getitimer（2），以便提供更好的计时精确度。
4. 当 alarm 信号消失后，函数将消息 Alarm Clock 打印到屏幕上。

sleep 函数。sleep 函数会让进程暂停一定的秒数，或者在没有指定秒数时永远停止该进程。读者可使用 alarm 函数来中断休眠的进程。

格式

```
sleep(SECONDS);
sleep SECONDS;
sleep;
```

示例 18.75

```
(The Script)
    #!/bin/perl
1   $|=1         # flush output buffer
2   alarm(5);
    print "Taking a snooze...\n";
3   sleep 100;
4   print "\07 Wake up now.!\n";

(Output)
2   Taking a snooze...    # Program pauses now for 5 seconds
4   (Beep) Wake up now.
```

解释

1. 变量 $| 强制在执行完读写操作后刷新缓冲区。
2. alarm 函数要求内核在 5 秒钟之内把信号 SIGALARM 发送给进程。
3. 进程休眠 100 秒，或者直接向它发送信号。
4. \07 表明在语句 Wake up now ！出现之前发出蜂鸣声。

请 Windows 用户注意！如需在 Win32 系统上使用 ActivePerl,则必须首先实现下列未完成的函数。其中最主要的函数是 alarm()，它只用在不多的几个 Perl 模块中。由于 ActivePerl 缺少其实现，因此用户将无法使用这些用到它的模块。下面是尚未实现的函数的完整列表：

有关进程和进程组的函数：

```
alarm(), fork(), getpgrp(), getppid(), getpriority(), setpgrp(),
setpriority()
```

有关获取用户和组信息的函数：

```
endgrent(), endpwent(), getgrent(), getgrgid(), getgrnam(),
getpwent(), getpwnam(), getpwuid(), setgrent(), setpwent()
```

有关系统 / 进程间通信的函数：

```
msgctl(), msgget(), msgrcv(), msgsnd(), semctl(), semget(),
semop(), shmctl(), shmget(), shmread(), shmwrite()
```

有关文件句柄、文件或目录的函数：

```
link(), symlink(), chroot()
```

有关输入和输出的函数：

```
syscall()
```

有关获取网络信息的函数：

```
getnetbyname(), getnetbyaddr(), getnetent(), getprotoent(),
getservent(), sethostent(), setnetent(), setprotoent(),
setservent(), endhostent(), endnetent(), endprotoent(),
endservent(), socketpair()
```

有关 ActivePerl 内建函数的可移植性问题，请读者参阅 perlport 和 perlwin32 上的相关文档页面。

18.6 读者应当学到的知识

1. 什么是系统调用？
2. Perl 是如何执行系统调用的？
3. 哪个模块能用于遍历文件系统？
4. 如何创建和删除目录？
5. 环境（environment）说明了什么？
6. 什么是进程？
7. fork 和 exec 函数实现了什么功能？
8. 在哪里可以找到处理 Windows 进程的模块？
9. 当需要执行系统命令时，使用系统函数和使用命令行替换（反引号）有哪些不同？
10. 什么是 globbing ？
11. eval 函数是如何与 die 函数配合工作的？

12. 什么是信号?
13. Perl 是如何处理信号的?
14. 如何在 Perl 中重命名文件?
15. 如何在 Perl 中删除一个文件。

18.7 下章简介

在下一章中，读者将学习如何在文本中插入 Perl 图像，以便让页面显示得更为友好。用户还将学习如何创建模板，以便根据模板规定的内容来确定打印时所用的字段宽度和数据类型。

此外，读者还将学习如何调整页面大小，如何打印页眉、页脚和汇总信息，以及如何把文本内容重定向到屏幕或另一个文件。

第 19 章　借助形象行编写报表

Perl 全称是实用摘要和报表语言。当用户在文件中实际提取并处理完数据之后，接下来就应当编写具有一定格式的报告，以便组织和总结这些信息。如果曾经用过 awk 编程语言编写报表的话，读者可能会在一开始觉得 Perl 的格式有点特别。

19.1　模板

为了编写一份报表，Perl 要求用户首先定义一个模板，以便描述如何显示这个报表；也就是说如何格式化该报表。譬如，需要将列左对齐、居中还是右对齐？有需要格式化的数字型数据吗？在每一页顶部需要标题或列标题吗？需要在报表末尾打印汇总数据吗？

本书将首先为一份简单报表提供一个简单的模板，然后再提供一个完整的示例。

格式模板（format template）的结构如下所示：

```
format FILEHANDLE=

    picture line
    value line (text to be formatted)

write;
```

19.1.1　定义模板的步骤

定义一个模板需要历经如下步骤：

1. 启动格式，首先使用关键字 format，随后是输出文件句柄名称和一个等号。默认的文件句柄是 STDOUT，即屏幕。模板定义可以位于脚本中的任何位置。

```
format FILEHANDLE=
```

2. 虽然模板中的文本内容会按原样打印出来，但模板一般都是由形象行（picture line）构成的，后者负责描述如何显示输出内容。形象（picture）由一组描述字段类型的符号（详见表 19-1）构成。这些字段可以是居中、左对齐或右对齐的（详见表 19-2）。形象行还可用于格式化数字值。

3. 在字段标志符（field designator）即 @ 符号之后，按照字段符号类型的字符数重复它。这就决定了字段的宽度。如果直接把 >>>>>> 放在 @ 符号之后的话，则它描述一个 7 字符宽的右对齐字

段⊖。是不是感到很奇怪？任何实际文本内容是不能直接放到 @ 符号之后的，否则就无法解释字段的类型了。

表 19-1　字段标志符

字段标志符	用途
@	表示字段的开始
@*	用于多行字段
^	用于填充字段

表 19-2　字段显示符号

<table>
<tr><th>字段符号类型</th><th>字段定义类型</th></tr>
<tr><td><</td><td>左对齐</td></tr>
<tr><td>></td><td>右对齐</td></tr>
<tr><td>|</td><td>居中</td></tr>
<tr><td>#</td><td>数字</td></tr>
<tr><td>.</td><td>指明小数点的位置</td></tr>
</table>

4. 形象行能将一行内容打断成多个字段。在形象行后面则是值行（value line），其内容将按照形象行的设定予以格式化。值行中的每个文本字段都由逗号隔开，并且与形象行中的字段符号一一对应。它会忽略值行中的任何空白字符。

5. 在创建好模板后，通过行上面的点号（.）结束模板定义。

6. 定义好模板后，使用 write 函数调用格式，并将格式化的记录发送到指定的输出文件句柄中。此时的默认文件句柄是 STDOUT。

示例 19.1

```
(The Script)
    #!/bin/perl
1   $name="Tommy";
    $age=25;
    $salary=50000.00;
    $now="03/14/97";
    # Format Template
2   format STDOUT=
3   ---------------------REPORT-------------------------
4   Name: @<<<<<< Age:@##Salary:@#####.## Date:@<<<<<<<<<<
5       $name,       $age,        $salary,       $now
6   .
         # End Template
7   write;
8   print "Thanks for coming. Bye.\n";

(Output)
    ---------------------REPORT-------------------------
    Name: Tommy   Age: 25 Salary: 50000.00 Date:03/14/97
    Thanks for coming. Bye.
```

解释

1. 为变量赋值。
2. 关键字 format 后面跟有 STDOUT，它是默认的也是当前选择的文件句柄。再后则是一个等号。
3. 按照原样打印该行内容。凡是没有专门予以格式化的文本，都将按照原样打印出来。
4. 该行又称作形象行（picture line）。该形象负责格式化输出内容。@ 定义了字段的开头。这

⊖ 虽然这里只有六个 > 符号，但 @ 字段标志符本身也算一个符号，因此字段宽度为 7 个字符。

里有四个字段，第一个字段是宽度为 7 个字符的左对齐字段，其前面是字符串 Name:，后面则是字符串 Age:；第二个字段由两个数字构成，随后是 Salary:；第三个字段由八位数构成，小数点插入到第 6 位之后，随后是字符串 Date。

5. 根据形象对这些变量进行格式化。每个变量都通过逗号隔开，并与上述形象字段一一对应。
6. 点号表明模板定义到此结束。
7. write 函数调用该模板，并将格式化后的输出内容显示到 STDOUT 上。

19.1.2 更改文件句柄

如果需要将报表写入文件而不是屏幕上，则应在打开文件时将其指定给一个文件句柄。在定义模板时，应当把同一个文件句柄用作格式文件句柄。若要调用格式，可以以输出文件句柄的名称作为参数调用 write 函数。

示例 19.2

```
    #!/bin/perl
1   $name="Tommy";
    $age=25;
    $salary=50000.00;
    $now="05/21/07";

2   open(REPORT, ">report" ) || die "report: $!\n";
    # REPORT filehandle is opened for writing
3   format REPORT=    # REPORT is also used for the format filehandle
    -----------------------
        | EMPLOYEE INFORMATION |
    -----------------------
4   Name: @<<<<<<
5       $name
    -----------------------
    Age:@###
        $age
    -----------------------
    Salary:@#####.##
        $salary
    -----------------------
    Date:@>>>>>>>>>>
        $now
    -----------------------
6   .
7   write REPORT;       # The write function sends output to the file
                        # associated with the REPORT filehandle
    -----------------------------------------------------------
(Output)
-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Tommy
-----------------------
Age:  25
-----------------------
Salary: 50000.00
```

```
-----------------------
Date: 05/21/07
-----------------------
```

解释

1. 定义变量。
2. 打开文件 report，并将它附加给文件句柄 REPORT。
3. 关键字 format 后面是输出文件句柄 REPORT。
4. 形象行描述了文本字符串 Name:，其后是一个宽度为 6 个字符的左对齐字段。
5. 按照相应形象行（见上一行）所描述的格式，对标量 $name 进行格式化。
6. 点号表明格式模板定义到此结束。
7. write 函数调用名叫 REPORT 的格式，并将格式化后的输出内容写到文件句柄中去。如果没有指定文件句柄，则文件句柄 REPORT 将不会接收到格式化的输出内容，因为本行并没有告诉 write 函数应当输出到哪里去。

19.1.3　页眉格式

在下面的示例中，会在每次调用格式时打印标题 EMPLOYEE INFORMATION。应该只在每一页的顶部打印该标题。Perl 允许用户定义页眉格式（top-of-the-page 格式），它只在开始新页面时调用。页面的默认长度为 60 行。打印 60 行之后，Perl 会在下一页的顶部打印出页眉格式（通过将特殊变量 $= 设置为其他值，便可改变默认的页面长度）。在下面的示例中，每当进入 while 循环体时，write 函数都会把所有输出内容发送到 STDOUT 上。

应用页眉格式之前的示例如下所示。

示例 19.3

```
(The File)
    $ cat datafile
    Tommy Tucker:55:500000:5/19/66
    Jack Sprat:44:45000:5/6/77
    Peter Piper:32:35000:4/12/93

(The Script)
    #!/bin/perl
1   open(DB, "datafile" ) || die "datafile: $!\n";
2   format STDOUT=
    -----------------------
    | EMPLOYEE INFORMATION |
    -----------------------
    Name: @<<<<<<<<<<<<
        $name
    -----------------------
    Age: @##
        $age
    -----------------------
    Salary: @#####.##
        $salary
    -----------------------
    Date: @>>>>>>>>>>
        $start
```

```
    .
3   while(<DB>){
        ($name, $age, $salary, $start)=split(":");
5       write ;
    }

(Output)
-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Tommy Tucker
-----------------------
Age: 55
-----------------------
Salary: 50000.00
-----------------------
Date: 5/19/66

-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Jack Sprat
-----------------------
Age: 44
-----------------------
Salary: 45000.00
-----------------------
Date: 5/6/77
-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Peter Piper
-----------------------
Age: 32
-----------------------
Salary: 35000.00
-----------------------
Date: 4/12/54
```

解释

1. 通过文件句柄 DB 以读方式打开文件 datafile。
2. 使用形象行和数据创建 STDOUT 的格式。
3. while 循环从 DB 文件句柄中一次读取一行内容。
4. 以冒号把每一行内容拆分为标量数组。
5. write 函数调用 STDOUT 格式，并将格式化的行发送到 STDOUT。

页眉格式如下所示：

格式

```
format STDOUT_TOP=

    picture line
    value line (text to be formatted)
```

```
.(End of template)
```

关键字 format 后面跟随着带有下划线和 TOP 的文件句柄名称。如果该行包含有形象行内容，则值行将由格式化后的文本构成。任何没有通过形象予以格式化的文本都将按照原样打印出来。点号表明页眉格式模板到此结束。

$% 是一个 Perl 特殊变量，含有当前的页码。

下面的示例显示了页眉格式的结果。

示例 19.4

```
(The Script)
    #!/bin/perl
    open(DB, "datafile" ) || die "datafile: $!\n";
    format STDOUT_TOP=
       -@||-
       $%
    -----------------------
    | EMPLOYEE INFORMATION |
    -----------------------
    .
    format STDOUT=

    Name: @<<<<<<<<<<<<<
        $name
    -----------------------
    Age: @##
        $age
    -----------------------
    Salary: @#####.##
        $salary
    -----------------------
    Date: @>>>>>>>
        $start
    .
    while(<DB>){
        ($name, $age, $salary, $start)=split(":");
        write;
    }


(Output)
          -  1 -
-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Tommy Tucker
-----------------------
Age: 55
-----------------------
Salary: 50000.00
-----------------------
Date: 5/19/66
```

```
Name: Jack Sprat
-----------------------
Age: 44
-----------------------
Salary: 45000.00
-----------------------
Date: 5/6/77

Name: Peter Piper
-----------------------
Age: 32
-----------------------
Salary: 35000.00
-----------------------
Date: 4/12/93
```

示例 19.5

```
(The Script)
    #!/bin/perl
1   open(DB, "datafile" ) || die "datafile: $!\n";
2   open(OUT, ">outfile" )|| die "outfile: $!\n";
3   format OUT_TOP=    # New filehandle
4       -@||-
5       $%
    -----------------------
    | EMPLOYEE INFORMATION |
    -----------------------
    .
    format OUT=
    Name: @<<<<<<<<<<<<<
        $name
    -----------------------
    Age: @##
        $age
    -----------------------
    Salary: @#####.##
        $salary
     -----------------------
    Date: @>>>>>>>
         $start
    -----------------------
    .
    while(<DB>){
        ($name, $age, $salary, $start)=split(":");
6       write OUT;
    }

(Output)
$ cat outfile
        -  1 -
-----------------------
/ EMPLOYEE INFORMATION /
-----------------------
Name: Tommy Tucker
-----------------------
Age: 55
```

```
-----------------------
Salary: 50000.00
-----------------------
Date: 5/19/66
-----------------------
Name: Jack Sprat
-----------------------
Age: 44
-----------------------
Salary: 45000.00
-----------------------
Date: 5/6/77
-----------------------
Name: Peter Pumpkin
-----------------------
Age: 32
-----------------------
Salary: 35000.00
-----------------------
Date: 4/12/93
-----------------------
```

19.1.4 select 函数

select 函数用于为 print 和 write 函数设置默认文件句柄。当使用 select 函数选定了某个文件句柄之后，print 和 write 函数就可以不再需要其他参数。在调用格式或调用 print 函数时，所选择的文件句柄将成为默认值。

select 函数会返回前面所选择的文件句柄的标量值。

如果有多种具有不同名称的格式，则可使用 $_ 变量来维护当前所选输出文件句柄的报表格式名称。write 和 print 函数会将其输出内容发送到当前选择的输出文件句柄中。

变量 $^ 含有用于当前所选输出文件句柄的页眉格式名称。

变量 $. 则含有记录数目（类似于 awk 中的 NR 变量）。

示例 19.6

```
(The Script)
    #!/usr/bin/perl
    # Write an awklike report
1   open(MYDB, "> mydb") || die "Can't open mydb: $!\n";
2   $oldfilehandle= select(MYDB);
                     # MYDB is selected as the filehandle for write
3   format MYDB_TOP =
                         DATEBOOK INFO

    Name           Phone          Birthday        Salary
    _____________________________________________________
    .

4   format MYDB =
    @<<<@<<<<<<<<<<<<<<<<@<<<<<<<<<<<<<@|||||||||@#######.##
    $.,      $name,         $phone,       $bd,      $sal
    .
```

```
5   format SUMMARY =
    _____________________________________________________
6   The average salary for all employees is $@######.##.
                                            $total/$count
    The number of lines left on the page is @###.
                                            $-
    The default page length is @###.
                               $=
7   .
    open(DB,"datebook") || die "Can't open datebook: $!\n";
    while(<DB>){
        ( $name, $phone, $address, $bd, $sal )=split(/:/);
8       write ;
        $count++;
        $total+=$sal;
    }
    close DB;

9   $~=SUMMARY;    # New report format for MYDB filehandle
10  write ;

11  select ($oldfilehandle); # STDOUT is now selected for further
                             # writes or prints
12  print "Report Submitted On" , 'date';

(Output)
16  Report Submitted On Sat Mar 26 11:52:04 PST 2001

(The Report)
    $ cat mydb
                          DATEBOOK INFO
    Name               Phone             Birthday      Salary
1   Betty Boop         245-836-8357      6/23/23       14500.00
2   Igor Chevsky       385-375-8395      6/18/68       23400.00
3   Norma Corder       397-857-2735      3/28/45       245700.00
           . . .
25  Paco Gutierrez     835-365-1284      2/28/53       123500.00
26  Ephram Hardy       293-259-5395      8/12/20       56700.00
27  James Ikeda        834-938-8376      12/1/38       45000.00
The average salary for all employees is $82572.50.
The number of lines left on the page is 32.
The default page length is 60.
```

解释

1. 以写方式打开 MYDB 文件句柄。
2. select 函数将 write 和 print 函数的默认文件句柄设置为文件句柄 MYDB。将标量 $oldfilehandle 赋值为前面已经指定的文件句柄。在这个示例中，前面定义的文件句柄就是默认的 STDOUT。
3. 定义文件句柄 MYDB 的页眉模板。
4. 将报表主题的格式设置为文件句柄 MYDB。
5. 使用新名字 SUMMARY 定义另一个格式模板。然后可以通过把格式名 SUMMARY 赋予特殊变量 $~ 的方式来调用该格式（见第 9 行）。
6. 定义形象行。
7. 结束格式模板。

8. 调用格式，将输出内容写入当前选择的 MYDB 文件句柄。
9. 将变量 $~ 赋值为新的格式名。将该格式应用到当前选择的 MYDB 文件句柄上。
10. write 函数调用当前选择的 MYDB 文件句柄的格式 SUMMARY。
11. select 函数将文件句柄设置为保存在变量 $oldfilehandle 中的值 STDOUT。此后，write 和 print 函数就会将其输出内容发送到 STDOUT 中，除非用户又选择了其他的输出文件句柄。
12. 将该行发送到屏幕。

19.1.5 多行字段

如果值行（value line）中拥有多行内容，则可使用 @* 变量启用多行字段，并在格式模板中将它放在单独的行上，然后给出多行值。

示例 19.7

```
(The Script)
    #!/bin/perl
1   $song="To market,\n
    to market, \nto buy a fat pig.\n";
2   format STDOUT=
3   @*
4   $song
5   @*
6   "\nHome again,\nHome again,\nJiggity, Jig!\n"
    .
    write;

(Output)
To market,
to market,
to buy a fat pig.
Home again,
Home again,
Jiggity, Jig!
```

解释

1. 标量 $song 含有换行符。
2. 将格式模板设置为 STDOUT。
3. @* 字段占位符（field holder）表明随后是一个多行字段。
4. 值行含有标量 $song，从而得到多行字符串。
5. @* 字段占位符（field holder）表明随后是一个多行字段。
6. 在值行中含有嵌有换行符的字符串。
7. 结束模板定义。

19.1.6 填充字段

脱字符（^）字段占位符允许用户创建一个含有文本的填充段落，其中文本是根据形象所定义的规范而放置的。如果文本在一行中填充不下，则会换行到下一行中，直到以段落块（paragraph block）的格式打印完所有行为止。文本的每一行内容都会拆分为多个词。Perl 会在指定行中放入

尽可能多的词。值行变量可以在多个行上重复，它只打印每一行剩余的文本，而不会再次打印整个值。如果值行数目多于实际要格式化的行数，则可能会出现空行。

读者可以通过使用特殊的～字符（又称作抑制指示符，即 suppression indicator）来抑制多余的空行。如果在值行中放置了两个连续的～，则待填充字段（以 ^ 开头）将持续不断地填充，直到所有文本都包含到段落中为止。

示例 19.8

```
(The Script)
    #!/bin/perl
    $name="Hamlet";
    print "What is your favorite line from Hamlet? ";
1   $quote = <STDIN>;
2   format STDOUT=
3   Play: @<<<<<<<<<<   Quotation:   ^<<<<<<<<<<<<<<<<<<
4         $name,                     $quote
5                                    ^<<<<<<<<<<<<<<<<<<
                                     $quote
                                     ^<<<<<<<<<<<<<<<<<<
                                     $quote
6   ~                                ^<<<<<<<<<<<<<<<<<<
                                     $quote
    .
    write;

(Output)
What is your favorite line from Hamlet? To be or not to be, that is the question:
Whether 'tis nobler in the mind to suffer the slings and arrows of outrageous fortune...

Play: Hamlet        Quotation: To be or not to be, that is the
                               question: Whether 'tis nobler in the
                               mind to suffer the slings and arrows
                               of outrageous fortune...
```

解释

1. 要求用户输入莎士比亚的 Hamlet 中的一行（允许换行）。
2. 定义 STDOUT 的格式模板。
3. 形象行中含有两个字段，其中之一是戏剧名称 $name 或 Hamlet，另一个则是用户输入行 $quote。字符占位符 ^ 用于创建填充段落。将摘录内容拆分为多个单词，每个词都填满 4 行内容。如果单词数目多于值行，则不进行格式化。如果单词数量小于值行，由于在最后一个形象行之前出现了 ~ 符号，因此需要抑制空白行出现。
4. 值行中含有根据上述形象完成格式化的变量。
5. 如果在第一行中放不下，$quote 的第二行会放在此处。
6. 如果在格式化三行以后就用完了文本，则在此抑制空白行出现。

示例 19.9

```
(The Script)
    #!/bin/perl
    $name="Hamlet";
    print "What is your favorite line from Hamlet? ";
```

```
    $quote = <STDIN>;
    format STDOUT=
    Play: @<<<<<<<<<<<  Quotation:   ^<<<<<<<<<<<<<<<<<<<
1          $name,                    $quote
2   ~~                               ^<<<<<<<<<<<<<<<<<<<
                                     $quote
    .
3   write;

(Output)
What is your favorite line from Hamlet? To be or not to be, that is the question:
Whether 'tis nobler in the mind to suffer the slings and arrows of outrageous fortune...

Play: Hamlet        Quotation: To be or not to be,
                               that is the
                               question: Whether
                               'tis nobler in the
                               mind to suffer the
                               slings and arrows of
                               outrageous fortune...
```

解释

1. 设置值行，其中含有戏剧名称与用户输入内容。
2. 使用两个～（抑制指示符）告诉 Perl 应当一直填入段，直到打印了 $quote 中的所有文本或者碰见空白行为止。
3. write 函数调用格式。

19.1.7 动态报表编写

在了解如何创建报表之后，读者还能根据需要在模板中动态地更改各个字段的宽度。首先，将指派的字段类型赋值给变量，并使用 Perl 的字符串重复运算符给出每个字段含有的字符数。根据字段宽度的不同，该字符数可以赋值为不同的数值。然后把报表模板赋值给一个字符串，该字符串中含有多个代表字段类型、字符数和字段值的变量。在需要时，用户可使用 Perl 的 eval 函数来解释格式字符串。详见示例 19.10。

示例 19.10

```
(The Script)
1  open(FH, "datebook") or die;  # Open a file for reading
2  open(SORT, "|sort") or die;   # Open a pipe to sort output

3  $field1="<" x 18;  # Create format strings
   $field2="<" x 12;
   $field3="|" x 10;
   $field4="#" x 6 . ".##";
   # Create the format template
4  $format=qq(
5  format SORT=
6    \@$field1\@$field2\@$field3\@$field4
7    \$name, \$phone, \$birth, \$sal
   .
   );
```

```
8  eval $format;
9  while(<FH>){
    ($name,$phone,$address,$birth,$sal)=split(":");
    ($first, $last)=split(" ", $name);
    $name=$last.", ". $first;
10  write SORT;
   }
    close(FH);
    close(SORT);

   (Output)
    Blenheim, Steve    238-923-7366  11/12/56    20300.00
    Boop, Betty        245-836-8357   6/23/23    14500.00
    Chevsky, Igor      385-375-8395   6/18/68    23400.00
    Corder, Norma      397-857-2735   3/28/45   245700.00
    Cowan, Jennifer    548-834-2348   10/1/35    58900.00
    DeLoach, Jon       408-253-3122   7/25/53    85100.00
    Evich, Karen       284-758-2857   7/25/53    85100.00
    Evich, Karen       284-758-2867   11/3/35    58200.00
    Evich, Karen       284-758-2867   11/3/35    58200.00
    Fardbarkle, Fred   674-843-1385   4/12/23   780900.00
    Fardbarkle, Fred   674-843-1385   4/12/23   780900.00
    Gortz, Lori        327-832-5728   10/2/65    35200.00
```

解释

1. 以只读方式打开文件 datebook。
2. 创建一个管道，以便把 Perl 的输出发送给系统提供的“排序”工具。该排序过程将按照姓氏的升序排列。
3. 创建变量，其中含有负责表达“形象行”的字符串。运算符“x”负责指定字符串重复次数。这么做使得用户可以方便地更改字段的宽度，从而能在显示报表时控制其外观。
4. 变量 $format 含有一个将用作报表格式的字符串。
5. 模板将通过第 2 行创建的管道把输出内容发送到 SORT 文件句柄中。
6. 这些变量表示每个字段的标志符及其宽度。请注意，这里在 @ 符号前面加上了一个反斜杠，以免 Perl 在条件成熟之前对该行求值。
7. 这些变量表示实际的字段值，这些值将基于上述形象行而显示出来。这些字段同样带有反斜杠，以免过早进行求值。
8. eval 函数负责对变量求值，并将得到的变量值替换回程序中。在解释每一行内容时，Perl 将会看到如下所示的当前报表格式：

```
                  format SORT=
@<<<<<<<<<<<<<<<<<<<@<<<<<<<<<<<<<@||||||||||@######.##
              $name, $phone, $birth, $sal.
```

9. 现在，每次一行地遍历文件，同时拆分其 name 字段以获得名字和姓氏，然后再把名字连接到姓氏后面，以便按照姓氏进行排序。
10. write 函数调用 SORT 模板，程序将按照第 6 行和第 7 行定义的报表模板将每一行内容写到屏幕上。

19.2 读者应当学到的知识

1. 读者应当能够定义出创建一个模板所需的步骤，模板负责生成格式化的输出。

2. 了解“形象行（picture line）”的意思。
3. 读者应当能够对页眉或页脚进行格式化。
4. 读者应当知道如何创建多行字段（multiline field）和填充字段（filled field）。
5. 能够更改格式文件句柄（format filehandle）。
6. 能够借助 eval 从头生成一个形象行。

19.3 下章简介

在下一章中，读者将学习 Perl 是如何与网络进行交互的，以及客户端 / 服务器模型。读者将学到什么是网络协议、网络寻址、Perl 的服务器与协议函数以及如何获取主机信息。此外还讨论了套接字的实现途径，以及如何使用它们在两台 Sun 计算机之间进行点对点通信。

练习 19 美观如画！

从下面的文本中生成一个报表，并将它发送到一个文件中。

```
+--------------+------------------------------------------+------------------------------+------------------+
|  supplierid  | companyname                              | contactname                  | phone            |
+--------------+------------------------------------------+------------------------------+------------------+
|            1 | Exotic Liquids                           | Charlotte Cooper             | (171) 555-2222   |
|            2 | New Orleans Cajun Delights               | Shelley Burke                | (100) 555-4822   |
|            3 | Grandma Kelly's Homestead                | Regina Murphy                | (313) 555-5735   |
|            4 | Tokyo Traders                            | Yoshi Nagase                 | (03) 3555-5011   |
|            5 | Cooperativa de Quesos 'Las Cabras'       | Antonio del Valle Saavedra   | (98) 598 76 54   |
|            6 | Mayumi's                                 | Mayumi Ohno                  | (06) 431-7877    |
|            7 | Pavlova, Ltd.                            | Ian Devling                  | (03) 444-2343    |
|            8 | Specialty Biscuits, Ltd.                 | Peter Wilson                 | (161) 555-4448   |
|            9 | PB KnSckebröd AB                         | Lars Peterson                | 031-987 65 43    |
|           10 | Refrescos Americanas LTDA                | Carlos Diaz                  | (11) 555 4640    |
|           11 | Heli Snwaren GmbH & Co. KG               | Petra Winkler                | (010) 9984510    |
|           12 | Plutzer LebensmittelgromSrkte AG         | Martin Bein                  | (069) 992755     |
|           13 | Nord-Ost-Fisch Handelsgesellschaft mbH   | Sven Petersen                | (04721) 8713     |
|           14 | Formaggi Fortini s.r.l.                  | Elio Rossi                   | (0544) 60323     |
|           15 | Norske Meierier                          | Beate Vileid                 | (0)2-953010      |
|           16 | Bigfoot Breweries                        | Cheryl Saylor                | (503) 555-9931   |
|           17 | Svensk Sjöföda AB                        | Michael Björn                | 08-123 45 67     |
|           18 | Aux joyeux ecclTsiastiques               | GuylFne Nodier               | (1) 03.83.00.68  |
|           19 | New England Seafood Cannery              | Robb Merchant                | (617) 555-3267   |
|           20 | Leka Trading                             | Chandra Leka                 | 555-8787         |
|           21 | Lyngbysild                               | Niels Petersen               | 43844108         |
|           22 | Zaanse Snoepfabriek                      | Dirk Luchte                  | (12345) 1212     |
|           23 | Karkki Oy                                | Anne Heikkonen               | (953) 10956      |
|           24 | G'day, Mate                              | Wendy Mackenzie              | (02) 555-5914    |
|           25 | Ma Maison                                | Jean-Guy Lauzon              | (514) 555-9022   |
|           26 | Pasta Buttini s.r.l.                     | Giovanni Giudici             | (089) 6547665    |
|           27 | Escargots Nouveaux                       | Marie Delamare               | 85.57.00.07      |
|           28 | Gai pGturage                             | Eliane Noz                   | 38.76.98.06      |
|           29 | ForOts d'Trables                         | Chantal Goulet               | (514) 555-2955   |
+--------------+--------------------------------------+------------------------------+------------------+
```

第 20 章　通过网络发送

20.1　网络与 Perl

由于在计算机之间共享信息和传输文件是做任何事情都不可或缺的，因此 Perl 也提供了很多函数，用于在程序中获取网络信息。为了编写用于进程间通信机制（套接字、消息队列等）的程序，了解一些网络相关的技术是非常必要的。下面讨论的内容只是介绍一些常用的网络术语，以便让读者在分析或编写含有这些函数的程序时，无需查阅 C 语言书籍或手册就能知道它们的意思。

20.2　客户端 / 服务器模型

在前面的章节中，当连接到某个数据库或 Web 服务器时，我们已经介绍过了客户端 / 服务器模型。大多数网络应用程序都使用了客户端 / 服务器模型。服务器可以为一个或多个客户端提供各种服务。客户端可以从位于本机或远端计算机上的服务器发送服务请求。服务器程序可以提供诸如电子邮件、Telnet 和 FTP 之类的服务。为了让客户端和服务器之间能够相互通信，必须在两个进程之间建立连接，这时通常都会用到套接字（socket）。目前最为著名的客户端 / 服务器模型莫过于 Web 使用的客户端（浏览器）/ 服务器（Web 服务器）模型了。

20.3　网络协议（TCP/IP）

在网络上发送数据时，必须有某种可靠的方式让数据从一台计算机传到另一台计算机上。为了完成这个复杂的过程，可以将网络组织为一系列的层（layer），其中每一层都会为下一层提供具体的网络服务。这些层相互独立，并具有明确定义的接口，负责为下一层提供功能。高层会把数据和信息传输到它下面的层，依此类推直到最底层。在最底层，两台计算机之间可以相互物理通信。负责在一台计算机上的网络层与另一台计算机上相应的网络层之间进行通信的规则和过程称为协议（protocol）。在 UNIX 中最常见的网络协议包括以太网（Ethernet）协议、IP、TCP 和 UDP。

20.3.1　以太网协议（硬件）

以太网层是网络的物理层。任何连接到以太网总线上的主机都能物理地访问网络上发送的数

据。以太网协议负责为经过线路的传输操作准备数据。它会把数据组织成帧（frame），这些帧含有 48 位的以太网源地址和目标地址。以太网层是网络的最底层，代表了物理网络上的数据传输。

20.3.2 Internet 协议（IP）

IP 层位于以太网层之上，负责为在独立网络上的路由操作准备数据。IP 使用长度为 4 个字节的 Internet 协议地址（即 IP 地址），并将数据组织成包（package）的信息片断。包中含有 IP 首部，带有源地址、目的地址、协议类型以及数据部分。虽然 IP 协议比以太网协议更为复杂，但它在传输包时却是无连接并且不可靠的。它不保证数据能顺利接收，但是如果待传输数据太大，IP 会将它们拆分为较小的单元。IP 协议对用户隐藏了不同网络之间底层细节上的差异。

20.3.3 传输控制协议（TCP）

虽然 IP 层提供了一点流控制能力，但不能保证其可靠性；也就是说，数据接收顺序可能会不同于发送顺序，或者数据根本无法到达目的地。TCP 协议则提供了可靠的端对端（end-to-end）服务，并且其流控制机制与电话呼叫相同。一旦建立了连接，两端便可相互通信；并且即便在同时对话时，其接收消息的顺序与发送顺序也是相同的。使用 TCP 协议的程序包括 rlogin、rsh、rcp 和 telnet 等。

20.3.4 用户数据报协议（UDP）

UDP 是一种能替换 TCP 的协议，用于不需要连续进行连接的应用程序，这些程序往往需要周期性地发送短消息，而不必考虑某些数据的丢失问题。UDP 数据报（datagram）类似于邮寄信件：每个包都有自己的地址，不保证递送的时间和顺序，并且允许发送重复的包[⊖]。这种方法避免了程序在等待结果时挂起。使用这种协议的程序包括 rwho 和 ruptime。

20.4 网络寻址

网络由很多相互连接的计算机构成，这些计算机又称作主机（host）。当把新计算机添加到网络上时，系统管理员会为它指定一个主机名。TCP/IP/Enternet 型网络上的每一台主机都拥有一个主机名和三种类型的地址：以太网日志、IP 地址和 TCP 服务端口号。当信息在网络中从一层传给另一层时，数据包应当含有相应的首部信息，其中包含了将包发送到下一个目标所需要的地址。

20.4.1 以太网地址

由于以太网地址在制造计算机时就已经烧入 PROM 中，因此它不是系统管理员分配的编号。它只用于标识某个硬件。/etc/ethers（UCB）文件含有面向某种特定网络的以太网地址和主机名（hostname）。

⊖ Rieken, B. and Weiman, L., *Adventures in UNIX, Network Applications Programming*，John Wiley & Sons, Wiley Professional Computing: New York, 1992, p.7.

20.4.2 IP 地址

IP 地址是一组 32 位的数字，由系统管理员负责分配给网络上的主机。如果一台主机同时连接到多个网络上，则必须对每个网络分别提供一个 IP 地址。该地址是由点号隔开的 4 个十进制数字组成的（如 129.150.28.56）。地址的第一部分标识了主机连接的网络，其余部分则代表主机。所有地址可以分成多个类（从 A 类到 C 类）。这些类决定了地址中哪些是属于网络的部分，哪些又是属于主机的部分。/etc/hosts 文件含有主机的地址、主机名以及与之关联的别名（详见 gethostent 函数及其相关函数）。

20.4.3 端口号

当服务多个用户进程时，在服务器上可能会有许多客户端对使用 TCP 和 UDP 协议的特定服务发送请求。当把信息传递到特定的应用层时，这些协议可使用 16 位的整数端口号来标识某个主机上的特定进程。0 到 255 之间的 TCP 和 UDP 端口号又称作通用端口，是为一些通用服务预留好的。（有的操作系统还为专用程序保留了特殊端口。）[⊖]最为常用的服务包括 Telnet 和 FTP，其 TCP 端口号分别是 23 和 21。如果编写使用 TCP 和 UDP 协议的应用程序，就必须为它分配惟一的端口号。该端口号应当是一个在保留端口号范围之外的数字。/etc/services 文件中含有通用端口号的列表。

20.4.4 Perl 协议函数

下面的 Perl 函数用于从 /etc/protocols 文件检索信息。这些函数都是根据 UNIX 手册页第 2、3 部分中相应的系统调用和库函数而命名的。

getprotoent 函数。getprotoent 函数会从网络协议数据库 /etc/protocols 中读取下一行内容并返回列表，其中的项目是协议的正式名称、协议的别名和替换名、还有协议号。setprotoent 函数则负责打开并回卷（rewind）文件 /etc/protocols。如果 STAYOPEN 值不是零，则在成功调用 getprotoent 函数之后不关闭该数据库。endprotoent 函数负责关闭数据库。

格式

```
getprotoent;
setprotoent (STAYOPEN);
endprotoent;
```

示例 20.1

```
(The Script)
1   while (($name,  $aliases, $proto ) = getprotent){
2   printf "name=%-5s,aliases=%-6sproto=%-8s\n",
            $name, $aliases, $proto;
    }

(Output)
2   name=ip,     aliases=IP    proto=0
```

⊖ Stevens, W. R., and Wright, G. R., *TCP/IP Illustrated, Volume 1: The Protocols*, Addison Wesley Longman, 1993, p.13.

```
    name=icmp,   aliases=ICMP  proto=1
    name=igmp,   aliases=OGMP  proto=2
    name=ggp,    aliases=GGP   proto=3
    name=tcp,    aliases=TCP   proto=6
    name=pup,    aliases=PUP   proto=12
    name=udp,    aliases=IDP   proto=17
```

解释

1. getprotoent 函数从 /etc/protocols 中获取项目内容。该循环将遍历读取整个文件。检索得到协议名、关联到该协议的任何别名以及协议号。
2. 打印每一项及其值。

getprotobyname 函数。getprotobyname 函数类似于 getprotoent 函数，也是从 /etc/protocols 文件中获取项。不同的是，getprotobyname 函数以协议名作为参数，返回其名称、别名以及协议号。

格式

```
getprotobyname(NAME);
```

示例 20.2

```
(The Script)
   ($name,  $aliases, $proto ) = getprotobyname('tcp');
   print "name=$name\taliases=$aliases\t$protocol number=$proto\n";

(Output)
name=tcp    aliases=TCP    protocol number=6
```

解释

从 /etc/protocols 文件中检索协议名称、别名和协议号。将协议名 TCP 作为 NAME 参数传递给该函数。

getprotobynumber 函数。getprotobynumber 函数类似于 getprotoent 函数，也是从 /etc/protocols 文件中获取项。不同的是，getprotobyname 函数以协议号作为参数，返回其名称、别名以及协议号。

格式

```
getprotobynumber(NUMBER);
```

示例 20.3

```
(The Script)
   ($name,  $aliases, $proto) = getprotobynumber(0);
   print "name=$name\taliases=$aliases\t$protocol number=$proto\n";

(Output)
name=ip    aliases=IP    protocol number=0
```

解释

getprotobynumber 函数以协议号作为参数，从 /etc/protocols 文件中检索协议名称、别名和协议号。

20.4.5 Perl 的服务器函数

下面的函数负责在网络服务文件 /etc/services 中查询信息。

getservent 函数。getservent 函数能从 /etc/services 文件中读取一行内容。如果 STAYOPEN 值不是零，则在每次调用之后都不关闭文件 /etc/services。

格式

```
getservent;
setservent (STAYOPEN);
endservent;
```

示例 20.4

```
(The Script)
1   setservent(1 );
2   ($name, $aliases, $port, $proto) = getservent;
3   print
"Name=$name\nAliases=$aliases\nPort=$port\nProtocol=$protocol\n";
            <  program continues here >

4   ($name, $aliases, $port, $proto) = getservent;
                     # Retrieves the next entry in /etc/services
5   print
"Name=$name\nAliases=$aliases\nPort=$port\nProtocol=$protocol\n";
6   endservent;

(Output)
3   Name=tcpmux
    Aliases=
    Port=1
    Protocol=tcp
5   Name=echo
    Aliases=
    Port=7
    Protocol=tcp
```

解释

1. setservent 函数确保在 STAYOPEN 标记非零时保持 /etc/services 文件为打开状态。
2. getservent 函数返回服务号、关联到服务的别名、端口号和网络协议。
3. 打印检索得到的值。
4. 第二次调用 getservent 函数，从 /etc/services 文件检索下一行内容。
5. 打印检索得到的值。
6. endservent 函数关闭网络文件。

getservbyname 函数。getservbyname 函数负责把服务端口名转译成相应的端口号。

格式

```
getservbyname(NAME, PROTOCOL);
```

示例 20.5

```
($name,$aliases,$port,$protocol)=getservbyname('telnet', 'tcp');
```

解释

服务名是 telnet，协议为 tcp, 返回服务名、别名、端口号和协议。

getservbyport 函数。getservbyport 函数也能从 /etc/services 文件中检索信息。

格式

```
getservbyport(PORT, PROTOCOL);
```

示例 20.6

```
(The Script)
    #!/bin/perl
    print "What is the port number? ";
    chomp($PORT=<>);

    print "What is the protocol? ";
    chomp($PROTOCOL=<>);

   ($name, $aliases, $port, $proto ) = getservbyport(
                                    $PORT, $PROTOCOL);
    print "The getservbyport function returns:
        name=$name
        aliases=$aliases
        port number=$port
        prototype=$protocol \n";

(Output)
What is the port number?  517
What is the protocol?  udp
The getservbyport function returns:
    name=talk
    aliases=talk
    port number=517
    prototype=udp
```

解释

将通用端口号 517 和协议名称 udp 传递给 getservbyport 函数，返回服务名、别名、端口号和协议名。

20.4.6 Perl 的主机信息函数

这些函数负责从 /etc/hosts 文件中检索信息。它们都是根据 UNIX man 页第 3 部分中相应的系统库例程而命名的。

gethostent 函数。 gethostent 函数能返回一个由 /etc/hosts 文件下一行内容所构成的列表。其中的项目包括主机的正式名称、主机地址的字节长度以及按字节顺序排列的网络地址列表。

格式

```
gethostent;
sethostent(STAYOPEN);
endhostent;
```

示例 20.7

```
(The Script)
    #!/bin/perl
```

```
1   while ( ($name,  $aliases, $addrtype, $length, @addrs) =
           gethostent ){
2       ($a, $b, $c, $d) = unpack ( 'C4', $addrs[0]);
3       print "The name of the host is $name.\n";
4       print "Local host address (unpacked) $a.$b.$c.$d\n";
    }

(Output)
3   The name of the host is localhost.
4   Local host address (unpacked) 127.0.0.1
    The name of the host is jody.
    Local host address (unpacked) 129.150.28.56
```

解释

1. gethostent 函数从 /etc/hosts 文件中检索下一行内容。
2. 将 gethostent 函数返回的原始地址解压为 4 个字节（C4），以便把它们打印出来。
3. 打印主机名 jody。
4. 打印本地主机的 IP 地址。

gethostbyaddr 函数。gethostbyaddr 函数会将一个网络地址转译为相应的主机名。它以一个原始地址作为参数，从 /etc/hosts 文件中检索相应的主机。其结果包括主机计算机的正式名称、主机别名或替换名的列表、返回地址的类型、以字节为单位的地址长度，以及按字节顺序排列的命名主机网络地址列表。

格式

```
gethostbyaddr(ADDRESS, DOMAIN_NUMBER);
```

示例 20.8

```
(The Script)
    #!/bin/perl
1   $address=pack("C4", 127,0,0,1);
2   ($name, $aliases,  $addrtype, $length, @addrs) = gethostbyaddr
        ($address,2);
3   ($a, $b, $c, $d) = unpack ( 'C4', $addrs[0]);
4   print "Hostname Is $name and the Internet address Is
        $a.$b.$c.$d.\n";

(Output)
Hostname is localhost and the Internet address is 127.0.0.1.
```

解释

1. Internet 地址是 127.0.0.1。将它压缩为 4 个字节，并用于 gethostbyaddr 函数中。
2. 将原始地址和 AF_INET（位于 /usr/include/sys/socket.h 中）的值传递给 gethostbyaddr 函数。
3. 将原始地址解压为 4 个字节的形式。
4. 打印主机号和 Internet 地址。

gethostbyname 函数。gethostbyname 函数能从 /etc/hosts 文件中返回一项内容，它以具体的主机名作为参数。其结果包括主机计算机的正式名称、主机别名或替换名的列表、返回地址的类型、以字节为单位的地址长度，以及按字节顺序排列的命名主机网络地址列表。

格式

```
gethostbyname(NAME);
```

示例 20.9

```
($name, $aliases, $addtrtype, $length,
        @addrs)=gethostbyname("dolphin");
```

解释

（参见上面的 gethostent 函数说明。）

20.5 套接字

套接字最初是由加利福尼亚大学伯克利分校（University of California Berkley）于 1982 年开发的，其目标是支持网络上的进程间通信。在进行进程间通信时，通过套接字用到了客户端 / 服务器模型。并不是任何操作系统都能支持套接字。如果读者所在系统不支持，则本节给出的套接字示例将无法运行。

套接字是一种代表了两个通信进程端点（即服务器和客户端）之间的软件抽象；也就是说，套接字使得进程之间能够实现对话。进行通信的进程可以位于同一台计算机上，也可在不同计算机上。服务器进程会创建一个套接字，客户端通过其名称获知这个套接字。为了确保网络上数据的来源地址和发送地址套接字用到了 IP 和端口地址。使用套接字的程序会打开套接字（类似于打开文件），一旦套接字顺利打开，便可进行 I/O 操作，并读写信息到通信的进程。当打开文件时，会返回一个文件描述符。而在打开套接字时，则会将套接字的一个描述符返回给服务器，另一个返回给客户端。套接字接口要比使用文件更为复杂，因为通信过程往往是跨网络的。

远程登录和文件传输程序是较为常见的使用套接字实现跨网络通信的实用程序。

20.5.1 套接字的类型

每个套接字都拥有通信路径的类型，用于标识如何通过该套接字传输数据。两种最为常见的套接字类型是 SOCK_STREAM 和 SOCK_DGRAM，二者分别使用了 TCP 和 UDP 协议。

还有一些不常用的套接字类型，例如 SOCK_SEQPACKET 和 SOCK_RAW，在这里将不予讨论它们。

流套接字（stream sockets）。 流套接字 SOCK_STREAM 类型提供了可靠的、面向连接的、序列化的双向服务。它提供了错误探测和流控制机制，并消除了重复的数据分段。其底层协议是 TCP。

数据报套接字（datagram sockets）。 数据报套接字 SOCK_DGRAM 类型提供了无连接的服务。它会以任何顺序接收发送的包，或者另一端根本收不到包。它不保证传递成功，并且包也可能出现重复。其底层协议是 UDP。

20.5.2 套接字域

在打开套接字时，会触发一些系统调用，它们描述了如何使用该套接字。进程必须规定通信的域（domain），又称作地址族（address family），它负责标识套接字命名的方式，并决定为发送和接

收数据需要使用哪一种协议和地址。套接字域负责标识服务器和客户端的位置：是在同一台计算机上，在 Internet 上，还是在 Xerox 网络上。

UNIX 域和 Internet 域是两种标准域。在使用套接字进行进程间通信时，UNIX 域负责同一台计算机上的进程通信，而 Internet 域则用于使用 TCP/IP 协议的远程计算机上的进程通信（AF_NS 域则用于处理 Xerox 网络）。

UNIX 域和 AF_UNIX 地址族。 如果是本地计算机上的两个进程间进行通信，其地址族（AF）就是 AF_UNIX。UNIX 域既支持 SOCK_STREAM 套接字类型，也支持 SOCK_DGRAM 套接字类型。UNIX 域上的 SOCK_STREAM 类型提供了进程间的双向通信能力，类似于实用程序 pipe。SOCK_STREAM 在进程间提供了可靠的字节流传输。数据报套接字则是不可靠的，因此不常用。此外，该套接字的行为就像是信息队列或电报一样。

UNIX 域中的套接字具有 UNIX 文件系统内的路径名。ls -l 命令列出了第一个字母为 s 的套接字文件（详见示例 20.10）。在系统文件 /usr/include/sys/socket.h 中，将 AF_UNIX 地址族赋值为常量 1。

示例 20.10

```
$ ls -lF greetings
srwxrwxrwx        1  ellie     0 Apr 26 09:31   greetings=
```

Internet 域和 AF_INET 地址族。Internet 套接字标识为 AF_INET，用于在不同计算机上的进程之间进行通信。Internet 域也支持套接字类型 SOCK_STREAM 和 SOCK_DGRAM。

Internet 套接字由负责标识 Internet 主机的 IP 地址和负责标识主机上端口的端口号定义。

SOCK_STREAM 使用的底层协议为 TCP。SOCK_DGRAM 使用的底层协议则是 UDP。在系统文件 /usr/include/sys/socket.h 中，将 AF_INET 地址族定义为常量 2。

套接字地址。在创建好套接字后，需要提供一个地址，以便通过该地址将数据发送给它。在 AF_UNIX 域中，这个地址就是一个文件名；而在 AF_INET 域中，则是 IP 地址和端口号。

20.5.3 创建套接字

函数 socket 会创建套接字，并返回一个指向该套接字的文件句柄。在服务器中，这个套接字又称作集结套接字（rendezvous socket）。

格式

```
socket(SOCKET_FILEHANDLE, DOMAIN, TYPE, PROTOCOL);
```

示例 20.11

```
    $AF_UNIX=1;
    $SOCK_STREAM=1;
    $PROTOCOL=0;

    socket(COMM_SOCKET, $AF_UNIX, $SOCK_STREAM, $PROTOCOL);
```

解释

在本示例中，socket 函数创建了文件句柄 COMM_SOCKET。其域或地址族是 UNIX，套接字类型是 STREAM 型，协议号则赋值为 0，以便允许系统为该套接字选择正确的协议。

20.5.4 绑定地址和套接字名

bind 函数。bind 函数负责将地址或名称附着到 SOCKET 文件句柄（参阅本章末尾的示例程序。）

格式

```
bind(SOCKET_FILEHANDLE, NAME);
```

示例 20.12

```
bind(COMM_SOCKET, "/home/jody/ellie/perl/tserv");
bind(COMM_SOCKET, $socket_address);
```

解释

1. 当使用 UNIX 域时，套接字文件句柄 COMM_SOCKET 将和 UNIX 路径名绑定。
2. 当使用 Internet 域时，套接字文件句柄 COMM_SOCKET 将绑定到该套接字类型所含地址上。

20.5.5 创建套接字队列

listen 函数。listen 函数负责在套接字上等待 accept，并规定拒绝进一步请求之前所允许的排队连接总数。设想在电话上已有 5 个呼叫者正在排队等候呼叫⊖。如果客户端请求数目超出了队列容量，则返回错误信息。如果调用成功，则返回真，否则返回假。其错误代码将保存为 $!。

格式

```
listen(SOCKET_FILEHANDLE, QUEUE_SIZE);
```

示例 20.13

```
listen(SERVERSOCKET, 5);
```

解释

队列中等待的最大连接数设置为 5。listen 函数会对到来的 accept 函数请求进行排队。

20.5.6 等待客户端请求

accept 函数。服务器进程中的 accept 函数会等待客户端请求的到达。如果队列中有请求，则在建立连接时从队列内删除第一个请求。然后，accept 函数会使用与原来的套接字或一般套接字（又称作集结套接字）相同的属性打开新的套接字文件句柄，并将它附加给客户端的套接字。新的套接字可以与客户端套接字通信。这时便可使用一般套接字来接受附加的连接了。如果调用成功，accept 函数会返回真，否则就返回假。其错误信息将保存为 $!。

格式

```
accept(NEWSOCKET, GENERICSOCKET);
```

⊖ Wall, L., and Schwartz, R. L., *Programming Perl*, 2nd ed., O' Reilly & Associates: Sebastopol, CA, 1998, p.158.

示例 20.14

```
accept (NEWSOCK, RENDEZ_SOCK );
```

解释

服务器进程使用 accept 函数接收来自客户端的请求。它会取队列中等待的第一个请求，并创建新的套接字文件句柄 NEWSOCK，其用到的属性和 RENDEZ_SOCK 文件句柄相同，后者又称作一般套接字或集结套接字。

20.5.7 建立套接字连接

connect 函数。connect 函数用到了套接字文件句柄和要连接到的服务器地址。在服务器上，它负责配合 accept 函数生成一个集结（rendezvous）。在建立连接之后，便可传输数据了。如果使用的是 UNIX 域，则提供 UNIX 文件路径名。如果使用的是 Internet 域，其地址则是对该服务器类型正确的网络地址。如果调用成功，该函数返回真，否则就返回假。其错误信息将保存在 $! 中。

格式

```
connect(SOCKET, ADDRESS);
```

示例 20.15

```
1  connect(CLIENT-SOCKET, "/home/joe/sock" );
2  connect(CLIENTSOCKET, $packed_address);
```

解释

1. 在 UNIX 域中，将一个 UNIX 路径名附加到文件句柄 CLIENTSOCKET 上。
2. 在 Internet 域中，将压缩后的网络地址附加到文件句柄 CLIENTSOCKET 上。该地址是由 gethostbyname 函数得到的，并经过 Perl 的 pack 函数予以压缩。例如：

```
$hostname="houston";
$port=9876;
$AF_INET=2;
$SOCK_STREAM=1;
($name, $aliases, $type, $len, $address)=gethostbyname($hostname);
$packed_address=pack(S n a4 x8, $AF_INET, $port, $address);
```

20.5.8 关闭套接字

shutdown 函数。shutdown 函数按规定关闭套接字连接。如果参数 HOW 值为 0，则在套接字上进一步的 receives 请求将遭拒绝。如果参数 HOW 值为 1，则套接字上的进一步 sends 请求将遭拒绝。如果参数 HOW 值为 2，则所有的 sends 和 receives 都将停止。

格式

```
shutdown(SOCKET, HOW);
```

示例 20.16

```
shutdown(COMM_SOCK, 2);
```

解释

套接字文件句柄 COMM_SOCK 禁止了进一步的 sends 和 receives，从而将它们关闭。

图 20-1 展示了一个有关套接字的示例。

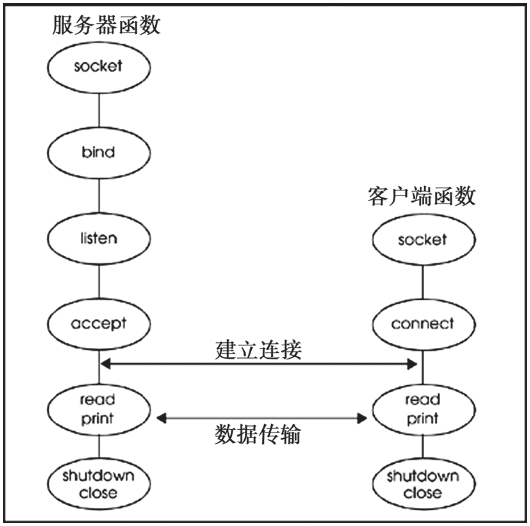


图 20-1　面向连接的服务器和客户端套接字

20.6　客户端 / 服务器程序

这里展示的示例程序已经在运行 SunOS 4.1.13 和 SunOS 5.3 的 Sun 工作站上测试通过。这些程序是 UNIX 域和 Internet 域的流套接字示例。虽然它们都支持数据报（UDP），但后者是不可靠的，并且不保证数据包能成功传递。rwho 和 ruptime 这样的程序一般都用在 Internet 域中，而很少用在 UNIX 域。

20.6.1　同一台计算机上面向连接的套接字

下面的脚本是一个简单的对通信端点使用流套接字的客户端 / 服务器模式示例。客户端和服务器都位于同一台计算机上，客户端从服务器读取并接收消息，将消息打印到屏幕上。示例中的错误检查和信号处理代码都较少。这些示例只是展示如何使用上面介绍的套接字函数。（有关本章未包含的套接字函数，请参阅附录 A 中的表 A.1。）

服务器程序

示例 20.17

```
(The Script)
    #!/bin/perl

    # The server and the client are on the same machine.
    print "Server Started.\n";
    $AF_UNIX=1;      # The domain is AF_UNIX
    $SOCK_STREAM=1;  # The type is SOCK_STREAM
    $PROTOCOL=0;     # Protocol 0 is accepted as the "correct
                     # protocol" by most systems.
```

```
4    socket(SERVERSOCKET, $AF_UNIX, $SOCK_STREAM, $PROTOCOL) ||
         die " Socket $!\n";
     print "socket OK\n";
5    $name="./greetings";  # The name of the socket is associated
                           # within the file system
     unlink "./greetings" || warn "$name: $!\n";

6    bind(SERVERSOCKET, $name) || die "Bind $!\n";
     print "bind OK\n";

7    listen(SERVERSOCKET, 5) || die "Listen $!\n";
     print "listen OK\n";

     while(1){

8    accept(NEWSOCKET, SERVERSOCKET ) || die "Accept $!\n";
     # Accept client connection

9    $pid=fork || die "Fork: $!\n";
10   if ($pid == 0 ){
11       print (NEWSOCKET "Greetings from your server!!\n";

12       close(NEWSOCKET);
         exit(0);
     }
     else{
13       close (NEWSOCKET);
     }
   }
```

解释

1. 将域设置成 UNIX。客户端和服务器位于同一台机器上。
2. 套接字的类型是 SOCK_STREAM，表示面向连接的字节流类型的通信。
3. protocol 设置为 0。如果该值为 0，则表示由系统负责处理。
4. 调用 socket 函数。在服务器上创建文件句柄 SERVERSOCKET。
5. 文件 greetings 是套接字 SERVERSOCKET 关联到的文件系统中的名称，是需要附加给套接字文件句柄的真实名称。如果该文件已经存在，则使用 unlink 函数删除它。
6. 套接字文件句柄与 UNIX 文件 greetings 绑定在一起。
7. listern 函数允许进程规定能从客户端接收多少个等待的连接。将这些请求进行排队，其最大排队数为 5。
8. accept 函数等待客户端请求并创建新的套接字文件句柄 NEWSOCKET，它与原先的 SERVERSOCKET 文件句柄（又称作集结套接字）拥有相同的属性。NEWSOCKET 是实际负责与客户端通信的套接字。集结套接字仍保持可用，以便接受后来的连接。
9. 使用 fork 函数创建子进程。这时，父进程和子进程都在运行。
10. 如果返回的 pid 是 0，说明子进程正在执行中。如果 pid 非零，说明父进程正在执行中。
11. 如果子进程正在执行中（即返回的 pid 为 0），将欢迎消息写入到套接字 NEWSOCKET 中。服务器与客户端进行通信。
12. 子进程关闭 NEWSOCKET 文件句柄并退出子进程。
13. 父进程关闭 NEWSOCKET 文件句柄，以便接受其他客户端请求。

客户端程序

示例 20.18

```
(The Script)
    #!/usr/bin/perl
    print "Hi I'm the client\n";

1   $AF_UNIX=1;
2   $SOCK_STREAM=1;
3   $PROTOCOL=0;

4   socket(CLIENTSOCKET, $AF_UNIX, $SOCK_STREAM, $PROTOCOL);
5   $name="./greetings";

    do{
       # Client connects  with server
6      $result = connect(CLIENTSOCKET, "$name" );
       if ($result != 1 ){
          sleep(1);

       }
7  }while($result !=  1 ); # Loop until a connection is made

8  read(CLIENTSOCKET, $buf, 500);
9  print STDOUT "$buf\n";
10 close (CLIENTSOCKET);
   exit(0);
```

解释

1. 这里的域是 AF_UNIX。客户端和服务器位于同一台计算机上。赋予标量的值就是系统文件 /usr/include/sys/socket.h 中赋给 AF_UNIX 宏的值。socket 函数以这些值作为参数创建套接字。
2. 套接字的类型是 SOCK_STREAM，是客户端和服务器之间一种有序、可靠、双向的通信方式。赋予标量的值就是系统文件 /usr/include/sys/socket.h 中赋给 AF_UNIX 宏的值。socket 函数以这些值作为参数创建套接字。
3. 协议值由包含在所创建套接字中的系统调用负责处理，它决定了如何在底层实现套接字。如果把协议值指定为 0，socket 函数就会认为它是“正确的协议”；即无需关心其协议细节。
4. socket 函数创建文件句柄 CLIENTSOCKET。
5. 把套接字文件句柄 CLIENTSOCKET 关联到 UNIX 文件 greetings。
6. connect 函数会把 CLIENTSOCKET 文件句柄连接到 UNIX 文件 greetings 上，使得客户端能够与服务器进行通信。如果调用成功，该函数会返回真，否则就返回假。
7. 程序不断循环执行，直到成功建立连接为止。
8. 在建立连接后，客户端从服务器套接字读取 500 个字节，并存储到标量 $buf 中。
9. 将 $buf 的内容打印到 STDOUT。
10. 关闭 CLIENTSOCKET 文件句柄。如果需要进行更多的控制，可使用 shutdown 函数。其结果如图 20-2 所示。

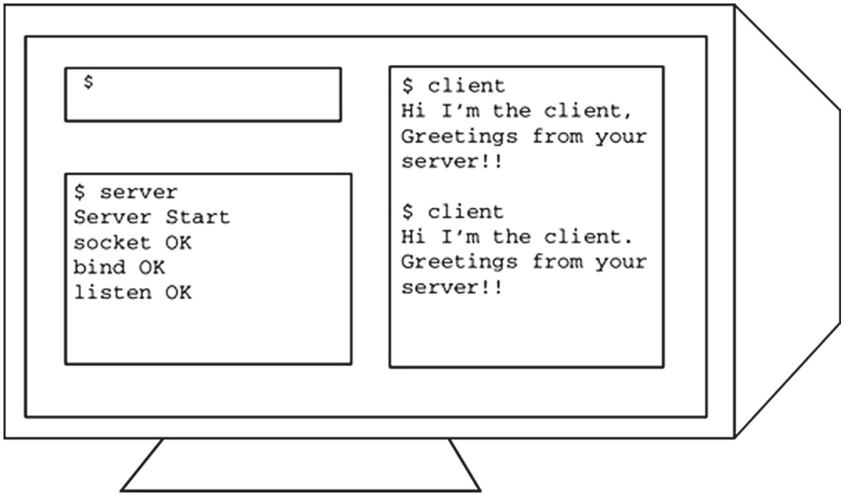


图 20-2 输出内容

20.6.2 远程计算机（Internet 客户端和服务器）上面向连接的套接字

下面的程序执行在两个 Sun 工作站上。其中，服务器的主机名是 scarecrow，运行了 SunOS 5.3；客户端的主机名则是 houston，运行 SunOS 4.1.3。在这个程序中，一台计算机上的客户端会向另一台计算机上的服务器请求当前时间。然后服务器把时间发送到客户端的套接字，并由客户端以可读的格式将它打印到屏幕上。这些示例都没有用到 Perl 5 标准库中的 Socket.pm 模块。其中所有的值都是硬编码的，因此无法从一台计算机移植到另一台计算机（详见 20.7 节）。

示例 20.19

```
#!/usr/bin/perl  -T
# timeserver -- a Time Server program,
# opens a Rendezvous Socket on port 9876
# and waits for a client to connect.
# When each client connects, this server determines the machine
# time on its host and writes the value on the communication
# socket to the client.
#
#                     Usage: timeserver [port number]
#
use strict;
use warnings;
($port)=@ARGV;
$port=9876 unless $port;
$AF_INET=2;
$SOCK_STREAM = 1;
$sockaddr = 'S n a4 x8';
($name,$aliases,$proto)=getprotobyname('tcp');
if($port !~ /^\d+$/){
    ($name, $aliases, $port)=getservbyport($port,'tcp');
}
print "Port = $port\n";
$this = pack($sockaddr, $AF_INET, $port, "\0\0\0\0");
select(COMM_SOCK); $| = 1; select (STDOUT);
    # Create R_SOCKET, the rendezvous socket descriptor
```

```
11 socket(R_SOCKET, $AF_INET, $SOCK_STREAM, $proto ) ||
          die "socket: $!\n";
     # Bind R_SOCKET to my address, $this
12 bind(R_SOCKET, $this) || die "bind: $!\n";
13 listen(R_SOCKET, 5) || die "connect: $!\n";
     # Infinite loop - wait until client connects,
     # then serve the client
   while(1){
14     accept(COMM_SOCK, R_SOCKET) || die "$!\n";
15     $now = time;
16     print COMM_SOCK $now;
   }
```

解释

1. 以命令行参数的形式传递主机名（见第 7 行）。
2. 如果 ARGV 数组为空数组，则为标量 $port 赋值 9876。该端口号是由程序员指定的，是一个预留端口号范围之外的数字。在 Sun 系统中，端口号 1024 是保留的。而端口号 9876 又称作是短暂端口（ephemeral port）。
3. 将标量 $AF_INET 赋值为 2，即系统文件 /usr/include/sys/socket.h 中赋给 AF_INET 宏的值。该数字代表 Internet 域，即 AF_INET。
4. 套接字类型是 SOCK_STREAM，它在系统文件 /usr/include/sys/socket.h 的对应值为 1。
5. pack 函数对套接字地址使用这种格式。
6. getprotobyname 函数以 tcp 协议作为名称返回正式的协议名、别名和协议号。
7. 如果没有给标量 $port 赋数字值，而是从命令行传入主机名的话，可使用 getservbyport 函数通过 tcp 协议返回正确的服务器端口号。
8. 打印端口号。
9. 将 Internet 域的地址和端口号压缩成一个二进制结构，该结构由无符号的短整数值构成，按照“network”顺序予以排列，包括 4 个 ASCII 字符和 8 个空值字节。在与之对比的 C 程序中，注意到它获取地址的方法是使用 sockaddr（见第 5 行）结构（详见 /usr/include/sys/socket.h）。Perl 负责处理大部分的工作。
10. 选择 socket 文件句柄作为当前默认的输出句柄。如果 $! 特殊变量设置为 1，则强制对每个 write 和 print 清除缓冲区。当把输出内容发送到终端时，stdout 文件句柄是按行缓冲的，其他则是按块缓冲的。当输出到管道或者套接字时，清除缓冲区。
11. socket 函数创建集结套接字文件句柄 R_SOCKET。
12. bind 函数将套接字文件句柄绑定到服务器的正确地址上。
13. listen 函数将队列上限设置为 5，即等待客户端请求的最大数目。
14. accept 函数等待客户端请求。当拿到请求时，通过创建新的 COMM_SOCKET 文件句柄来接受它。该文件句柄的所有属性与 R_SOCK ET 相同。COMM_SOCK 是负责与客户端通信的服务器套接字。
15. time 函数返回从 UTC 时间 1970 年 1 月 1 日开始计算的非闰年秒数。
16. 将时间值发送给套接字文件句柄 COMM_SOCK。

示例 20.20

```
#!/usr/local/bin/perl
# timeclient--a client for the Time Server program,
# creates a socket and connects it to the server on port 9876.
```

```
   # The client then expects the server to write the server's
   # host time onto the socket. The client simply does
   # a read on its socket, SOCK, to get the server's time.
   #
   #           Usage: timeclient [server_host_name]
   #
   print "Hi, I'm in Perl program \'client\' \n";
1  ($them) = @ARGV;
2  $them = 'localhost' unless $them;
3  $port = 9876 ;      # timeserver is at this port number
4  $AF_INET = 2;
5  $SOCK_STREAM = 1;
6  $sockaddr = 'S n a4 x8';
7  ($name, $aliases, $proto) = getprotobyname('tcp');
8  ($name,$aliases, $port, $proto)=getservbyname($port, 'tcp')
       unless $port =~ /^\d+$/;

9  ($name,$aliases, $type, $len, $thataddr)=gethostbyname($them);
10 $that = pack($sockaddr, $AF_INET, $port, $thataddr);
   # Make the socket filehandle
11 if ( socket(SOCK, $AF_INET, $SOCK_STREAM, $proto ) ){
       print "Socket ok.\n";
   }
   else { die $!; }
   # Call up the server
12 if(connect(SOCK, $that)){
       print "Connect ok.\n";
   }
   else { die $!;}
   # Set socket to be command buffered
13 select(SOCK); $| = 1; select (STDOUT);
   # Now we're connected to the server, let's read her host time
14 $hertime = <SOCK>;
   close(SOCK);
   print "Server machine time is: $hertime\n";
15 @now = localtime($hertime);
   print "\t$now[2]:$now[1] ", $now[4]+1,"/$now[3]/$now[5]\n";
```

解释

1. 服务器的主机名可以作为命令行参数予以传递。
2. 如果 ARGV 数组为空数组，则把主机名设置为 localhost。
3. 为了识别服务器进程，客户端需要知道服务器的端口号。
4. 域是 Internet。
5. 套接字类型是 SOCK_STREAM，它在 /usr/include/sys/socket.h 的赋值为 1。
6. pack 函数对套接字地址使用这种格式。
7. getprotobyname 函数以 tcp 协议作为名称返回正式的协议名、别名和协议号。
8. getservbyname 函数返回服务器的正式名称、别名、端口号和协议名，除非 $port 已经含有指定的数字。
9. 通过 gethostbyname 函数从主机获取原始地址信息。
10. 将 Internet 域的地址和端口号压缩成一个二进制结构，该结构由无符号的短整数值构成，按照“network”顺序予以排列，包括 4 个 ASCII 字符和 8 个空值字节。在与之对比的 C 程序中，注意到它获取地址的方法是使用 sockaddr 结构。
11. socket 函数会创建一个 Internet 域的、面向连接的套接字文件句柄 SOCK。
12. connect 函数将客户端的套接字连接到服务器地址上。

13. 选择 SOCK 文件句柄，打印并在写入完毕后清空缓冲区。
14. Perl 从 SOCK 文件句柄中读取内容，并检索服务器时间。
15. 将时间转换为本地时间，并予以打印。其结果如图 20-3 所示。

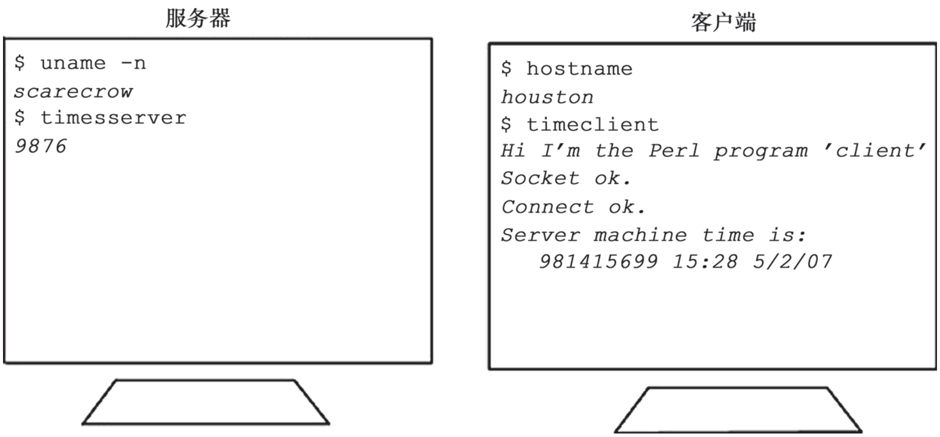


图 20-3　远程计算机上面向对象的套接字

20.7　Socket.pm 模块

虽然套接字本来是 Berkley 为 UNIX 系统开发的，但现在很多操作系统都能支持它。Perl 5 引入了一个特殊的 Socket.pm 模块来处理套接字，使程序可以很容易地从一台计算机输出到另一台计算机上。因为机器要求的必备函数和常量都在该模块中处理，因此还可以避免像上面的示例那样把值硬编码到程序中。不过，读者应当小心：在使用前请务必理解套接字的工作方式。其常量名往往很不直观，而且 Perl 5 之后的版本还在模块中引入了更多的功能。

下面所示图 20-4 展示了如何利用 Socket.pm 和 Perl 5 提供的其他程序编写前面介绍的 TCP/IP 服务器和客户端程序，从而让程序在网络上使用时变得更为安全。

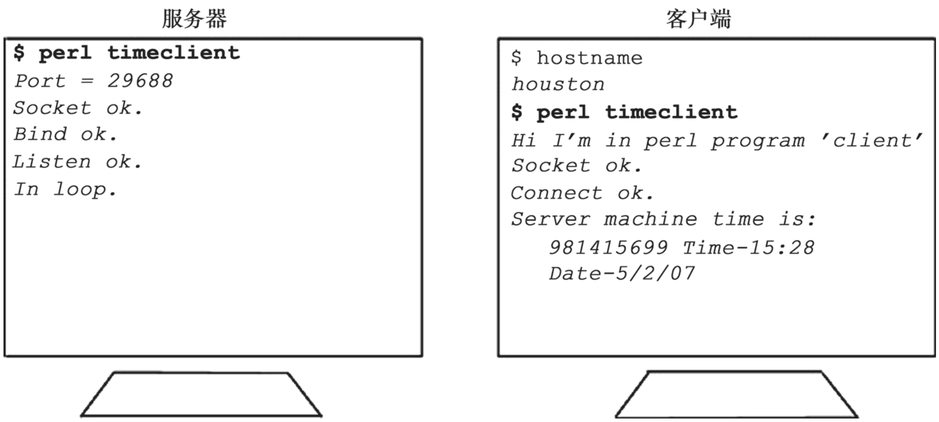


图 20-4　服务器 / 客户端和 Socket.pm，下面的示例将讨论它们

服务器

示例 20.21

```
    (The Server Script)
    1   #!/bin/perl  -Tw
    2   require 5.6;
    3   use strict;
    4   use Socket;
    5   use FileHandle;

        # timeserver --  a Time Server program, opens a rendezvous
        # socket on port 29688 and waits for a client to connect.
        # When each client connects, this server determines the machine
        # time on its host and writes the value on the communication
        # socket to the client.
        #
        #                      Usage: timeserver
        #
    6   my($this, $now);
    7   my $port = shift || 29688;

    8   $this = pack('Sna4x8', AF_INET, $port, "\0\0\0\0");
        print "Port = $port\n";
    9   my $prototype = getprotobyname('tcp');
    10  socket(SOCKET, PF_INET, SOCK_STREAM, $prototype) ||
                              die "socket: $!\n";
        print "Socket ok.\n";

    11  bind(SOCKET, $this) || die "bind: $!\n";
        print "Bind ok.\n";

    12  listen(SOCKET, SOMAXCONN) || die "connect: $!\n";
        print "Listen ok.\n";

    13  COMM_SOCKET->autoflush;
        SOCKET->autoflush;

        # Infinite loop -- wait until client connects,
        # then serve the client
    14  while(1){
            print "In loop.\n";
    15      accept(COMM_SOCKET, SOCKET) || die "$!\n";
            print "Accept ok.\n";
    16      $now = time;
    17      print COMM_SOCKET $now;
        }
```

解释

1. 当遇到只使用一次标识符、在设置之前使用标量、以非数字值作为编号等情况时，-w 开关将发送诊断信息到 STDERR。-T 开关会打开污染检查，以免传入的数据影响到程序内部逻辑。如果使用了目录路径，则污染模式还可检查其他人是否有权写目录，并检查传入程序的参数。如果被污染，则数据（或引用污染数据的任何变量）将不能用到调用子 shell 的命令中。建议读者对服务器程序和 CGI 脚本（即其他人会运行的程序）打开污染检查开关。

2. require 以版本号作为参数。这样能避免使用 5.6 之前版本的 Perl。
3. strict 编译指示确保程序不会使用不安全的结构。它禁止了符号引用、裸字，且必须使用 my 来声明变量。
4. 该程序使用 Socket 模块。该模块的设计目标是让套接字变得更加易于移植。
5. 在这里使用 FileHandle 模块，以便使用其 autoflush 方法。该方法会强制在写套接字之后清除缓冲区。
6. 这些变量将在后面的程序中用到。必须使用 my 声明它们，因此 strict 强制要求这一点，以提供安全特性。
7. 读者可以将主机名作为命令行参数进行传递，并转送到变量 $port 中。如果 ARGV 数组为空数组，则将标量 $port 赋值为 9688，这是保留端口号范围之外的一个数字。
8. AF_INET 是在 Socket 模块中定义的代表 Internet 域的常量。pack 函数会把服务器的 IP 地址和端口号压缩到 $this 中。
9. getprotobyname 函数以 tcp 协议作为名称返回正式的协议名、别名和协议号。
10. socket 函数创建集结套接字文件句柄 SOCKET。
11. bind 函数将套接字文件句柄绑定到服务器的正确地址上。
12. listen 函数会设置队列的 SOMAXCONN 限制，即等待客户端请求的最大数量（一般是 5）。
13. FileHandle 模块的 autoflush 方法会强制在写套接字之后清除缓冲区。
14. 开始无穷循环。现在，服务器会等待客户端的请求。
15. accept 函数等待客户端请求。当得到请求时，通过创建新的 COMM_SOCK 文件句柄来接受它。该文件句柄的所有属性与 SOCKET 相同。
16. 服务器调用 time 函数获得当前时间，并将它赋值给 $now。
17. 服务器将时间发送到 COMM_SOCK。客户端在终端获得它。

客户端

示例 20.22

```
    #!/usr/local/bin/perl -Tw
    require 5.6.0;
1   use Socket;
    use FileHandle;
2   use strict;
3   my($remote, $port, @thataddr, $that,$them, $proto,@now,
        $hertime);

    # timeclient --  a client for the Time Server program,
    # creates a socket and connects it to the server on
    # port 29688.
    # The client then expects the server to write server's
    # host time onto the socket, so the client simply does
    # a read on its socket, SOCK, to get the server's time
    #
    #
    #                  Usage:  timeclient [server_host_name]
    #
    print "Hi, I'm in perl program \'client\' \n";
4   $remote = shift || 'localhost' ;
```

```
5    $port = 29688 ;  # timeserver is at this port number
6    @thataddr=gethostbyname($remote);

7    $that = pack('Sna4x8', AF_INET, $port, $thataddr[4]);

8    $proto = getprotobyname('tcp');

     # Make the socket filehandle

9    if ( socket(SOCK, PF_INET, SOCK_STREAM, $proto ) ){
         print "Socket ok.\n";
     }
     else { die $!; }

     # Call up the server
10   if (connect(SOCK, $that)) {
         print "Connect ok.\n";
     }
     else { die $!;}

     # Set socket to be command buffered
11   SOCK->autoflush;

     # Now we're connected to the server, let's read her host time
12   $hertime = <SOCK>;
13   close(SOCK);

14   print "Server machine time is: $hertime\n";
15   @now = localtime($hertime);
16   print "\tTime-$now[2]:$now[1] ",
            "Date-",$now[4]+1,"/$now[3]/$now[5]\n";

(Output)
$ perl timeserver
Port = 29688
Socket ok.
Bind ok.
Listen ok.
In loop.

$ perl timeclient
Hi, I'm in perl program 'client'
Socket ok.
Connect ok.
Server machine time is: 981415699
        Time-15:28 Date-2/5/07
```

解释

1. 该程序使用 Socket 模块。
2. 使用 strict 编译指示确保该程序用到的变量都是“安全的”。
3. 这些变量将在后面的程序中用到。必须使用 my 声明它们，因此 strict 强制要求这一点，以提供安全特性。
4. 将主机名作为命令行参数进行传递，并转送到变量 $port 中。如果 ARGV 数组为空数组，则把 $port 标量赋值为 localhost。

5. 如果已经赋了值，则由客户端获取服务器的端口号。
6. 客户端在这里获取服务器的正式地址。使用 gethostbyname 函数获得原始地址信息。
7. 将 Internet 域的地址和端口号压缩成一个二进制结构，该结构由无符号的短整数值构成，按照“network”顺序予以排列，包括 4 个 ASCII 字符和 8 个空值字节。
8. 返回 tcp 协议信息。
9. socket 函数创建一个 Internet 域的、面向连接的套接字文件句柄 SOCK。
10. connect 函数将客户端套接字绑定到服务器地址上。
11. autoflush 方法会在打印和写入后强制清除缓冲区。
12. Perl 从 SOCK 文件句柄读取内容。并通过套接字检索服务器的时间。
13. 关闭套接字。
14. 从服务器检索时间值（从 UTC 时间 1970 年 1 月 1 日开始计算的非闰年秒数）并打印它。
15. 将该时间转换为本地时间，并赋值给数组 @now。
16. 打印转换后的时间。

20.8 读者应当学到的知识

1. 什么是 TCP ？
2. 什么是 Perl 协议函数？
3. 端口号的功能是什么？
4. 什么服务会用到端口号 21 ？
5. 以太网地址的用途是什么？
6. 什么是套接字？
7. 流套接字和数据报套接字的区别是什么？
8. 什么是套接字的域？
9. accept 函数的功能是什么？它位于套接字的服务器一侧还是客户端一侧？
10. 使用 Socket.pm 模块的好处是什么？

附录 A　Perl 内建函数、编译指示、模块和调试器

A.1　Perl 函数

下面的表 A-1 详细列出了所有 Perl 函数，并简述了这些函数的功能。请注意 ：位于括号中的文本是 UNIX 手册页第 2 部分中与之同名的 UNIX 系统调用参考信息。与之同名的 UNIX 库函数则位于 UNIX 手册页的第 3 部分。

表 A-1　相等性运算符与数字值

函　数	描　述
abs	abs VALUE 返回其参数（默认是 $_）的绝对值，忽略其正负符号
accept	accept(NEWSOCKET, GENERICSOCKET) 接收来自正在等待连接的客户端的一个套接字连接 GENERICSOCKET 是一个文件句柄，由 socket 函数负责事先打开它、绑定它到地址、并监听该连接。NEWSOCKET 也是一个文件句柄，其属性与 GENERICSOCKET 相同。accept 函数会把 GENERICSOCKET 附加到刚刚创建的连接上 参见 accept(2)
alarm	alarm(SECONDS) alarm SECONDS 在 SECONDS 秒之后，将 SIGALARM 信号发送给指定进程 参见 alarm(3)
atan2	atan2(X,Y) 返回范围为 <pi> 的 X/Y 的 arctangent 值
bind	bind(SOCKET，NAME) 将地址 NAME 绑定到已经打开的匿名套接字 SOCKET 上 参见 bind(2)
binmode	binmode(FILEHANDLE) binmode FILEHANDLE 用在区分文本模式与“二进制”模式的操作系统（不是 UNIX）。为以二进制模式读取数据而做准备
bless	bless(REFERENCE, CLASS) bless REFERENCE 表明 REFERENCE 所引用的对象位于指定包（CLASS）中; 如果没有给出 CLASS 参数，则表示在当前包中。该函数将返回引用

（续）

函　数	描　述
caller	caller(EXPR) caller EXPR caller 返回一个含有子例程调用堆栈信息的数组，其中含有包名、文件名、行号等。如果 EXPR 是数字的话，该函数将向后查找当前帧之前的第 EXPR 个堆栈帧
chdir	chdir (EXPR) chdir EXPR chdir 将当前工作目录切换到 EXPR。如果省略了 EXPR 的话，则会切换到该用户的主目录 参见 chdir(2)
chmod	chmod (MODE, LIST) chmod MODE, LIST 更改列表中文件的权限：第一个参数是权限 MODE 号（八进制）；另一个参数则是文件名的列表。该函数将返回修改成功的文件总数 参见 chmod(2)
chomp	chop (LIST) chop (VARIABLE) chop VARIABLE chop 删去字符串 VARIABLE 的末字符；或者当字符对应于 $_ 当前值时，删去 LIST 中每一项的末字符。与 chop（如下所述）函数不同，该函数将返回删去的字符总数
chop	chomp (LIST) chomp (VARIABLE) chomp VARIABLE chomp 删去字符串 VARIABLE 的末字符，或删去 LIST 中每一项的末字符，并返回删去的字符值。当没有提供参数时，删去 $_ 的末字符
chown	chown (LIST) chown LIST 更改列表中文件的拥有者和组 ID，列表前两个元素分别是数值形式的 uid 和 gid。列表中其余元素则是文件名。返回更改成功的文件数目 参见 chown(2)
chr	chr NUMBER 返回 NUMBER 的 ASCII 码值。例如，chr(66) 将返回 B
chroot	chroot (FILENAME) chroot FILENAME 将当前进程的根目录更改为 FILENAME，它是一个以 / 开头的路径名起始点。必须以超级用户身份才能执行该函数 参见 chroot(2)
close	close (FILEHANDLE) close FILEHANDLE 关闭 FILEHANDLE 关联的文件、套接字或管道
closedir	closedir (DIRHANDLE) closedir DIRHANDLE 关闭由 opendir 打开的目录结构 参见 directory(3)
connect	connect(SOCKET, NAME) 把一个进程连接到另一个等待 accept 调用的进程上。其中 NAME 是压缩后的网络地址 参见 connect(2)
cos	cos (EXPR) cos EXPR 以弧度格式返回 EXPR 的 cosine（余弦）值

（续）

函　数	描　述
crypt	crypt (PLAINTEXT, SALT） 口令加密函数，其中 PLAINTEXT 是用户口令，SALT 则是由字符集 [a-zA-Z] 中的字符所构成的双字符字符串 参见 crypt(3)
dbmclose	dbmclose(%ASSOC_ARRAY) dbmclose %ASSOC_ARRAY 断开 DBM 文件及其相关数组之间的绑定。如果支持的话，它将只适用于 NDBM，它是新版的 DBM 格式 参见 untie 参见 dbm(3)
dbmopen	dbmopen(%ASSOC_ARRAY, DBNAME, MODE) 把一个 DBM 或 NDBM 文件绑定到一个关联的数组上。在访问数据库之前，必须先通过 dbmopen 函数打开它。文件 file.dir 和 file.pag 必须存在。 DBNAME 是不含扩展名 .dir 或 .pag 的文件名。如果数据库不存在并指定了权限 MODE 的话，则创建该数据库 参见 tie 参见 dbminit(3)
defined	defined (EXPR) defined EXPR 如果 EXPR 是一个实际的值，则返回布尔数 1。如果 EXPR 不含实际值，则返回布尔数 0。EXPR 可以是标量、数组、散列或子例程。对于散列，该函数只会检查它是否定义了值（键）
delete	delete $ASSOC{KEY} 从关联数组中删除指定的值。如果删除成功，则返回删除的值；否则返回未定义的值。如果删除 %ENV 中的值，便会改变环境变量。亦可使用 undef 函数，后者更为快捷
die	die (LIST) die LIST die 将 LIST 打印到 STDERR，并以 $! 的值（即系统错误消息 errno）退出。当位于 eval 函数中时，该函数会把 $@ 设置为错误消息，并退出 eval 块。如果 LIST 的值不以换行符结束的话，则在消息后面附加当前的脚本、行号或换行符
do	do BLOCK do SUBROUTINE(LIST) do EXPR do BLOCK 会返回 BLOCK 块中最后一个命令的值。do SUBROUTINE(LIST) 则会调用定义的 SUBROUTINE 子例程。do EXPR 会将 EXPR 用作文件名，并把该文件当作 Perl 脚本来执行。它们主要用于导入 Perl 子例程库中的子例程
dump	dump LABEL 对某个直接二进制核心镜像（immediate binary image core）进行转存。undump 命令则用于解除转存，它不是 Perl 5.6.0 的组件
each	each (%ASSOC_ARRAY) each %ASSOC_ARRAY 随机返回一个二元数组，含有关联数组（散列）的下一个值的键和值
eof	eof(FILEHANDLE) eof() eof 如果从 FILEHANDLE 中读取到的下一个字符是文件末尾的话，该函数就会返回 1。如果省略了 FILEHANDLE，则返回上一次读取的文件末尾情况
eval	eval (EXPR) eval EXPR Perl 程序会在当前 Perl 脚本的上下文语境中判断 EXPR。该函数常用于捕获其他致命错误。负责把语法错误、运行时错误和来自 die 函数的错误消息返回给变量 $@。如果没有错误的话，则将变量 $@ 设置为 NULL。其返回值是上一次求值的表达式结果值

（续）

函　数	描　述
exec	exec (LIST) exec LIST 在当前程序的上下文语境中执行系统命令 LIST。该函数不会返回任何内容。如果 LIST 是一个标量，则查找 shell 元字符，并将其传递给 /bin/sh。否则，将参数传递给 C 函数调用 execvp。该函数不会清除缓冲区
exists	exists EXPR 如果关联数组中存在指定的键，则返回 TRUE，即使没有与之对应的值也是如此
exit	exit (INTEGER) exit INTEGER 以状态值 INTEGER 退出脚本。如果省略 INTEGER 的话，则以 0 退出，也就是说程序以成功状态退出。非零状态意味着程序出错
exp	exp (EXPR) exp EXPR 指数函数。返回 e 的 EXPR 次幂值
fcntl	fcntl(FILEHANDLE, FUNCTION, SCALAR) 修改所打开文件的属性。需要用到 sys/cntl.ph。FUNCTION 可以重复现有的文件描述符，获取或设置文件描述符标记，获取或设置文件状态标记，获取或设置异步 I/O 所有权，获取或设置记录锁。SCALAR 则是一个整数型的标记 参见 fcntl(2)
fileno	fileno (FILEHANDLE) fileno FILEHANDLE 返回整数形式的 FILEHANDLE 文件描述符。该描述符的值 0、1、2 分别表示 STDIN、STDOUT、STDERR。如果在文件关闭后再次打开的话，则该函数在 Perl 脚本中就可能是不可靠的 参见 fileno(3)
flock	flock (FILEHANDLE, OPERATION) 在文件上加上或者取消锁。OPERATION 负责规定对文件的加锁、共享锁、独占锁或非独占锁的操作。删除某个文件的 OPERATION 则是 unlock 参见 flock(2)
fork	fork 创建一个新的（子）进程。子进程是父进程的一个副本。子进程和父进程都会继续执行 fork 后面的指令。该函数将把 0 返回给子例程，并将子进程的 pid 返回给父进程
format	format NAME = 　picture line 　value list 　... 负责声明一组形象（picture）行，用以描述相应值的布局。write 函数将使用规定的格式把输出发送到 NAME 代表的命名文件句柄上。如果省略了 NAME，默认的文件句柄则为 STDOUT
formline	formline PICTURE, LIST format 函数用到的内部函数，负责格式化形象行中对应值的列表。读者也可从程序中直接调用该函数
getc	getc (FILEHANDLE) getc FILEHANDLE getc 从关联到 FILEHANDLE 上的输入中返回下一个字符。如果碰到 EOF，则返回 NULL。如果省略了 FILEHANDLE 的话，则从 STDIN 中读取
getgrent	getgrent setgrent endgrent 反复遍历 /etc/group 文件，并从中返回各个列表项，包括组名、口令、组 ID（gid）和成员 参见 getgrent(3)

（续）

<table>
<tr><th>函　数</th><th>描　述</th></tr>
<tr><td>getgrgid</td><td>getgrgid(GID)
根据组的编号返回组文件列表项
参见 getgrgid (3)</td></tr>
<tr><td>getgrnam</td><td>getgrnam (NAME)
根据组名返回组文件列表项
参见 getgrnam (3)</td></tr>
<tr><td>gethostbyaddr</td><td>gethostbyaddr(ADDRESS, AF_INET)
将网络地址翻译为对应的名称和替换地址。该函数会返回主机名、别名、地址类型、长度以及未包装的原始地址。AF_INET 的值始终是 2
参见 gethostbyaddr(3)</td></tr>
<tr><td>gethostbyname</td><td>gethostbyname(HOSTNAME)
将主机名翻译成 /etc/hosts 中的列表项，包括主机名、别名和地址。在标量上下文语境中，该函数将只返回主机的地址
参见 gethostbyname (3)</td></tr>
<tr><td>gethostent</td><td>gethostent
sethostent(STAYOPEN)
endhostent
反复遍历 /etc/hosts 文件，并从中返回各个列表项，包括名称、别名、地址类型、长度和替换地址。该函数负责从网络数据库 /etc/hosts 中返回列表内容
参见 gethostent (3)</td></tr>
<tr><td>getlogin</td><td>getlogin
如果存在 /etc/utmp 文件的话，从该文件中返回当前的登录情况。如果 getlogin 不起作用，请读者尝试：
$loginname = getlogin || (getpwuid($<))[0] || die "Not a user here"
参见 getlogin (3)</td></tr>
<tr><td>getnetbyaddr</td><td>getnetbyaddr(ADDR, ADDRESSTYPE)
将网络地址翻译成其对应的网络名。该函数能从网络数据库 /etc/networks 中返回列表内容。在标量型上下文语境中，该函数将只返回网络名
参见 getnetbyaddr(3)</td></tr>
<tr><td>getnetbyname</td><td>getnetbyname (NAME)
将网络名翻译成其对应的网络地址。该函数能从网络数据库 /etc/networks 中返回列表内容。在标量型上下文语境中，该函数将只返回网络地址
参见 getnetbyname (3)</td></tr>
<tr><td>getnetent</td><td>getnetent
setnetent(STAYOPEN)
endnetent
反复遍历 /etc/networks 文件，并从中返回各个列表项。该函数负责从网络数据库 /etc/networks 中返回列表内容。在标量上下文语境中，该函数将只返回网络地址
参见 getnetent (3)</td></tr>
<tr><td>getpeername</td><td>getpeername(SOCKET)
返回套接字连接的另一端所包装的 sockaddr 地址
参见 getpeername (2)</td></tr>
<tr><td>getpgrp</td><td>getpgrp(PID)
getpgrp PID
返回给定 PID 对应的当前进程组（PID 0 是当前进程）。如果没有 EXPR 参数的话，该函数将返回当前进程所在的进程组
参见 getpgrp (2)</td></tr>
<tr><td>getppid</td><td>getppid
返回父进程的 pid。如果返回值为 1，就表明是 init 进程的 pid。Init 会接纳那些没有父进程的进程
参见 getppid (2)</td></tr>
</table>

（续）

函　数	描　述
getprioity	getprioity (WHICH, WHO) 返回 WHICH 的值，即当前进程、进程组或用户的优先权（nice）值。如果 WHO 的值为 0，则表示当前的进程、进程组或用户 参见 getprioity (2)
getprotobyname	getprotobyname (NAME) 将协议名 NAME 翻译成对应的协议号，并返回一个包含协议名、别名和协议号的列表。该函数将从网络协议数据库 /etc/protocols 中返回一行内容 参见 getprotobyname (3)
getprotobynumber	getprotobynumber (NUMBER) 将协议号 NUMBER 翻译成对应的协议名，并返回一个包含协议名、别名和协议号的列表。该函数将从网络协议数据库 /etc/protocols 中返回一行内容 参见 getprotobynumber (3)
getprotoent	getprotoent setprotoent (STAYOPEN) endprotoent 反复遍历 /etc/protocols 文件，并从中返回各个列表项，包括协议名、别名和协议号。如果 STAYOPEN 参数值非零,则在随后的调用期间不关闭数据库。函数 endprotoent 负责关闭该文件。在标量型上下文语境中，该函数将只返回协议名 参见 getprotoent (3)
getpwent	getpwent setpwent endpwent 反复遍历 /etc/passwd 文件，并从中返回各个列表项，包括用户名、口令、uid、gid、磁盘配额、注释、gcos 字段、主目录和启动 shell 等。函数 endpwent 负责关闭该文件。在标量型上下文语境中，该函数将只返回用户名 参见 getpwent (3)
getpwnam	getpwnam(NAME) 将用户名转译为 /etc/passwd 文件中的对应项。该函数返回的列表内容包括用户名、口令、uid、gid、磁盘配额、注释、gcos 字段、主目录和启动 shell 等。在标量型上下文语境中，该函数将只返回用户 ID 参见 getpwent (3)
getpwuid	getpwuid (UID) 将用户 ID 转译为 /etc/passwd 文件中的对应项。该函数返回的列表内容包括用户名、口令、uid、gid、磁盘配额、注释、gcos 字段、主目录和启动 shell 等。在标量型上下文语境中，该函数将只返回用户名 参见 getpwent (3)
getservbyname	getservbyname(NAME, PROTOCOL) 从 /etc/services 数据库中，将端口名转译为对应的端口号，并返回一个含有服务名、别名、服务端口和所需协议的数组。在标量型上下文语境中，该函数将只返回服务的端口号 参见 getservent(3)
getservbyport	getservbyport (PORT_NUMBER, PROTOCOL) 从 /etc/services 数据库中，将端口号转译为对应的标量型端口名，并返回一个含有服务名、别名、服务端口和所需协议的数组。在标量型上下文语境中，该函数将只返回服务的端口名 参见 getservent(3)
getservent	getservent setservent(STAYOPEN) endservent 反复遍历 /etc/services 文件，并从中返回服务名、别名、服务端口和所需协议。如果 STAYOPEN 参数值非零，则在随后的调用期间将不关闭数据库。endservent 负责关闭该文件。在标量型上下文语境中，该函数将只返回服务的端口名 参见 getserv (3)

（续）

函　数	描　述
getsockname	getsockname(SOCKET) 返回 SOCKET 连接本地端的 sockaddr 地址 参见 getsockname(2)
getsockopt	getsockopt (SOCKET, LEVEL, OPTNAME) 返回请求选项 OPTNAME，该选项将在指定协议 LEVEL 上关联到 SOCKET 参见 getsockopt(2)
glob	glob EXPR 像 shell 一样对 EXPR 执行文件名扩展。若省略了 EXPR，则默认会使用 $_。也可使用内部运算符 <*>
gmtime	gmtime(EXPR) gmtime EXPR 将 time 函数得到的结果转换为一个含有格林威治时间值的 9 元数组，其中包含秒、分、时、日、月、年、周日、年日等信息，如果是夏令时的话，其中还会出现 1 参见 Perl 库模块 Time::Local 中的 ctime(3) 和 timegm() 函数
goto	goto LABEL goto EXPR goto &NAME 将程序转到 LABEL 处并继续执行。该函数不能 goto 到任何需要初始化的结构内，譬如 foreach 循环的子例程。goto 不会返回任何值。格式 goto &NAME 将当前运行的子例程替换为针对 NAME 的调用（常用于 AUTOLOAD 子例程中）
grep	grep (EXPR, LIST) grep BLOCK, LIST 把 LIST 中与 EXPR 匹配的任意元素返回给新数组。该函数将返回一个标量，即匹配成功的项目总数
hex	hex (EXPR) hex EXPR 将十六进制值 EXPR 返回为相应的十六进制字符串。如果没有 EXPR 的话，则默认使用 $_
import	import CLASSNAME LIST import CLASSNAME 这不是内建函数，而是模块规定的类方法，负责通过 use 函数将名字导出到其他模块中去
index	index(STR, SUBSTR, POSITION) index(STR, SUBSTR) 返回 STR 中 SUBSTR 的第一个匹配值的位置。POSITION 表示该串内从 0 开始计算的子串起始位置
int	int(EXPR) int EXPR 返回 EXPR 的整数部分。如果没有提供 EXPR 的话，则默认使用 $_
ioctl	ioctl (FILEHANDLE, FUNCTION, SCALAR) 用于控制 I/O 操作，主要是负责结束 I/O。要求用到 sys/ioctl.ph。此处的 FUNCTION 是一个 I/O 请求。SCALAR 将根据该请求进行读或者写 参见 ioctl(2)
join	join (EXPR, LIST) 将 LIST 中各个串连接在一起，返回单个串。其字段分隔符由定界符 EXPR 规定

（续）

函　数	描　述
keys	keys(%ASSOC_ARRAY) keys %ASSOC_ARRAY 返回一个由关联数组中所有的键所组成的数组
kill	kill(SIGNAL, PROCCESS_LIST） kiss PROCESS_LIST 把信号 SIGNAL 发送到列表中的各个进程。SIGNAL 可以是数字或者信号名(信号名必须位于引号中)。如果 SIGNAL 为负，则表示取消该进程组 参见 kill(2)
last	last LABEL last last 命令兼容于 C 里面的 break 命令。它负责退出最内层循环。如果带有循环标记 last LABEL 的话，则能退出整个循环
lc	lc EXPR 返回 EXPR 的小写形式。等效于 \L\E 转义序列
lcfirst	lcfirst EXPR 返回首字母小写的 EXPR。等效于 \l\E 转义序列
length	length (EXPR) length EXPR 返回标量 EXPR 的字符串长度。如果省略了 EXPR 的话，则默认会返回 $_ 的长度。该函数不能用于获取数组或关联数组的大小
link	link(OLDFILE, NEWFILE) 创建一个硬链接。NEWFILE 将成为 OLDFILE 的另一个名称 参见 link(2)
listen	listen(SOCKET, QUEUESIZE) 在 SOCKET 套接字上监听连接。QUEUESIZE 规定了等待连接的最大进程数 参见 listen(2)
local	local(LIST) 针对代码块、子例程或 eval 对 LIST 中的变量进行本地化
localtime	localtime (EXPR) localtime EXPR 将 time 函数返回的时间值转换为一个含有本地时区的 9 元素数组。该数组含有： 秒 分 时 日 月份编号（0 表示 1 月） 自 1900 年开始的年份 周日（0 表示星期天） 年日（0 表示 1 月 1 日） isdst（如果使用夏令时的话，该值为真） 参见 ctime(3)
lock	lock THING 在变量、子例程或 THING 引用的对象上面加锁，直到锁超出其作用域为止。如果启用它，则只能用在线程上
log	log(EXPR) log EXPR 返回 EXPR 的对数值（以 e 为底数）。如果省略 EXPR 的话，则默认求 log($_)

（续）

函　数	描　述
lstat	lstat (FILEHANDLE) lstat FILEHANDLE lstat (EXPR) 返回一个由符号链接上文件统计值所构成的 14 元素数组，而不是符号链接指向的文件本身。该数组含有： 设备 文件模式 指向文件的硬链接数目 拥有者的用户 ID 拥有者的组 ID 原始设备 文件大小 文件上一次访问时间 文件上一次修改时间 文件上一次状态改变的时间 文件系统实际分配的 I/O 首选块数大小 参见 stat(2)
m	/PATTERN/ m/ PATTERN/ m 是一个匹配运算符，负责将 PATTERN 解释为正则表达式，用在需要替换界定符的时候，譬如 m!PATTERN!
map	map (BLOCK, LIST) map (EXPR, LIST) 对于 LIST 中的每一个元素，计算其 BLOCK 或 EXPR 的值，并返回含有计算结果的列表值。下面的示例将把数字列表转译为字符：@chars = map chr, @numbers
mkdir	mkdir(NAME, MODE) 使用 MODE 权限（八进制）创建目录 NAME 参见 mkdir(2)
msgctl	msgctl(MSGID, CMD, FLAGS) msgctl 系统调用，用于针对消息队列的操作。它会产生较为特殊的返回码。要求用到库文件 ipc.ph 和 msg.ph。 参见 System V IPC 参见 msgctl(2)
msgget	msgget(KEY, FLAGS) 调用 msgget 系统调用。返回消息队列 ID 号；如果尚未定义的话，则返回错误信息 参见 System V IPC 参见 msgget (2)
msgrcv	msgrcv (MSGID, VAR, MSG_SIZE, TYPE, FLAGS) 调用 msgrcv 系统调用。从消息队列中检索消息，并将它存储在 VAR 中。MSG_SIZE 是消息队列的最大容量，TYPE 则表示消息的类型 参见 System V IPC 参见 msgrcv (2)
msgsnd	msgsnd (ID, MSG, FLAGS) 调用 msgsnd 系统调用。把消息 MSG 发送到消息队列中。MSG 必须以消息类型开头。用户可使用 pack 函数来创建消息 参见 System V IPC 参见 msgsnd (2)

（续）

函　数	描　述
my	my TYPE EXPR; ATTRIBUTES my EXPR; ATTRIBUTES my TYPE EXPR my EXPR 使用 my 函数声明的变量将成为专有变量，即它们将只能存储在最内层的包含块、子例程、eval 或文件中。只有简单标量、完整数组和散列才能使用 my 声明。这里的 TYPE 和 ATTRIBUTES 参数都是可选的
new	new CLASSNAME LIST new CLASSNAME 这不是内建函数，而是一个构造函数方法。它是由 CLASSNAME 模板定义的，负责创建 CLASSNAME 类型的对象。其创建规则取自 C++
next	next LABEL next 开始最内层循环，或者开始标记为 LABEL 的循环。等效于 C 语言中的 continue 函数
no	no Module LIST 如果已经使用 use 导入了编译指示（pragma）或模块的话，则可使用 no 函数表明不再使用它们
not	not EXPR 逻辑否定 EXPR 的真值
oct	oct (EXPR) oct EXPR oct 返回八进制字符串 EXPR 的十进制值。如果 EXPR 含有前导零的话，则将 EXPR 解释为十六进制。如果没有提供 EXPR 的话，则默认转换 $_
open	open (FILEHANDLE, EXPR) open (FILEHANDLE) open FILEHANDLE 打开实际文件 EXPR，并将它附加给文件句柄 FILEHANDLE。如果没有提供 EXPR 的话，则必须把与 FILEHANDLE 同名的标量值赋予该文件名 读　“FILEHANDLE” 写　“>FILEHANDLE” 读 / 写　“+>FILEHANDLE” 追加　“>>FILEHANDLE” 导出　“| UNIX Command” 导入　“UNIX Command |”
opendir	opendir (DIRHANDLE, EXPR) 打开名叫 EXPR 的目录结构，并在负责检查该结构的函数中把它附加给 DIRHANDLE 句柄 参见 directory(3)
ord	ord(EXPR) ord 返回 EXPR 中第一个字符所对应的无符号 ASCII 数值
our	our TYPE EXPR: ATTRIBUTES our EXPR: ATTRIBUTES our TYPE EXPR our EXPR 声明一个或多个在块、文件或 eval 中全局有效的变量。该函数类似于 my 或 global，但是它不会创建新的专用变量。当打开 strict 指示并且需要全局变量时，便可使用该函数

（续）

函　数	描　述
pack	$packed = pack(TEMPLATE, LIST) 负责将列表值包装成一个二进制结构，并返回该结构。其中 TEMPLATE 是带引号的含有数值或类型的字符串。TEMPLATE 可以是： a　ASCII 串，填充 null A　ASCII 串，填充空格 b　比特串，从低到高的顺序 B　比特串，从高到低的顺序 h　十六进制串，低 4 位在前 H　十六进制串，高 4 位在前 c　有符号的字符值 C　无符号的字符值 s　有符号的短整型值 S　无符号的短整型值 i　有符号的整数值 I　无符号的整数值 l　有符号的长整型值 L　无符号的长整型值 n　以“network”顺序出现的短整型值 N　以“network”顺序出现的长整型值 f　本地格式的单精度浮点数 d　本地格式的双精度浮点数 p　指向字符串的指针 x　空字节 X　备份字节 @　以空值填充的绝对精度 u　uuencode 编码字符串
package	package NAMESPACE 把包声明为 NAMESPACE，并创建单独的命名空间（符号表）。NAMESPACE 属于其他的最内层代码块、子例程、eval 或文件。如果包声明位于同一层的话，则新的声明将覆盖旧的声明
pipe	pipe (READHANDLE, WRITEHANDLE) 打开管道读或者写，一般都在 fork 后调用 参见 pipe(2)
pop	pop (ARRAY) pop ARRAY 取出数组中的一个元素。数组中的元素将减少一个
pos	pos (SCALAR) pos SCALAR 返回在 SCALAR 中搜索上一个匹配项之后字符的偏移量，即下一次搜索的起始位置。其偏移量从 0 起算。如果 $scalar 赋值为“hello”，并且搜索模式为 %scalar=~m/l/g 的话，pos 函数就会返回找到第一个 1 之后的字符位置，即 3
print	print(FILEHANDLE LIST) print(LIST) print FILEHANDLE LIST print LIST print 将字符串或以逗号隔开的字符串列表打印到 FILEHANDLE 上。这里可以打印到用户选择的 FILEHANDLE，也可打印到默认的 STDOUT。如果调用成功，该函数会返回 1，否则就返回 0
printf	printf(FILEHANDLE FORMAT, LIST) printf(FORMAT , LIST) 把格式化之后的字符串打印到 FILEHANDLE 上。如果省略了 FILEHANDLE 的话，则打印到当前已经选定的文件句柄上。其默认句柄为 STDOUT。该函数类似于 C 语言中的 printf，但是它不支持 * 参见 printf(3)

（续）

函　数	描　述
prototype	prototype FUNCTION 返回函数原型字符串，其中 FUNCTION 表示函数名。如果没有原型的话，则返回 undef
push	push (ARRAY, LIST) 将 LIST 中的值追加到 ARRAY 的尾部。该数组将会变大。该函数能返回 ARRAY 的新长度值
q, qq, qw, qx	q /STRING/ qq /STRING/ qw /LIST/ qx /COMMAND/ 引号的替换形式。q 结构会把 STRING 当作是位于单引号中一样处理。qq 结构会把 STRING 当作是位于双引号中一样处理。qw 结构会把 LIST 中的每个元素都当作是位于单引号中一样来处理。而 qx 则会把 COMMAND 当作位于反引号中一样来处理
quotemeta	quotemeta EXPR 返回 EXPR 的标量值，并给所有正则表达式元字符加上反斜杠
rand	rand (EXPR) rand EXPR rand 返回一个介于 0 和 EXPR 之间的随机分数值（标量）。其中 EXPR 是正数。如果不使用 srand 的话，该函数将生成相同的数字序列。如果省略 EXPR，则会返回介于 0 到 1 之间的值
read	read(FILEHANDLE, SCALAR, LENGTH, OFFSET) read(FILEHANDLE, SCALAR, LENGTH) 从 FILEHANDLE 中的 OFFSET 位置开始读取总数为 LENGTH 的字节内容，将其内容存入 SCALAR 中，并返回读取的字节总数。如果碰到 EOF 的话，则返回 0（类似于 fread 系统调用） 参见 fread(3)
readdir	readdir(DIRHANDLE) readdir DIRHANDLE 从 opendir 函数打开的目录结构 DIRHANDLE 中读取下一项 参见 directory(3)
readline	readline FILEHANDLE 从选定的 FILEHANDLE 中读取并返回一行内容。例如，$line=readline(STDIN)
readlink	readlink (EXPR) readlink EXPR 返回符号链接的值。其中 EXPR 是符号链接的路径名，如果省略的话则默认使用 $_ 参见 readlink(2)
readpipe	readpipe scalar EXPR readpipe LIST(proposed) 一个实现命令替换的 qw// 引号结构或反引号的内部函数。例如，为了打印 UNIX ls 命令的输出内容，可键入 print readpipe(ls)
recv	recv (SOCKET, SCALAR, LEN, FLAGS) 在套接字上将 LEN 字节长的信息接收到变量 SCALAR 中，并返回发送者的地址 参见 recv(2)
redo	redo LABEL redo 不重新判断条件值，立即重新执行循环体。如果没有可供执行的代码块，则不去执行它。如果没有提供 LABEL 的话，则重新开始执行最内层的循环
ref	ref EXPR 如果引用了 EXPR 的话，返回 EXPR 的数据类型，否则返回 NULL。该函数的返回值取决于引用的类型：REF、SCALAR、ARRAY、HASH、CODE 或 GLOB。如果 EXPR 是归属于某个包的对象的话，则它的返回值将是包（类）的名字
rename	rename(OLDNAME, NEWNAME) 将文件 OLDNAME 改名为 NEWNAME。如果跨越了文件系统的话，该函数将不起作用。如果 NEWNAME 已经存在的话，该函数将自动将它移除 参见 rename(2)

（续）

函　数	描　述
require	require (EXPR) require EXPR require 通过在 @INC 数组中搜索指定的文件，导入 Perl 库中的 EXPR。也可用于检查是否已经包含了指定的库。如果省略 EXPR 的话，则默认使用 $_
reset	reset (EXPR) reset EXPR reset 清除变量或数组。如果省略了 EXPR 的话，则重置 ?? 搜索
return	return LIST 从子例程中返回一个值。该函数不能用在子例程外面
reverse	reverse (LIST) 反转 LIST 的顺序，并返回得到的数组
rewinddir	rewinddir(DIRHANDLE) rewinddir DIRHANDLE 将 DIRHANDLE 的位置回卷到目录结构的开始位置。 参见 directory(3)
rindex	rindex(STRING, SUBSTR, OFFSET) rindex(STRING, SUBSTR) 如果给定了 OFFSET，该函数就会从 STRING 中返回由 OFFSET 处开始的最后一次 SUBSTR 出现的位置。它与 index 函数类似，不过后者将返回第一个位置，而非最后一个位置
rmdir	rmdir (FILENAME) rmdir FILENAME 删除空目录 FILENAME
s	s/SEARCH_PATTERN/REPLACEMENT/[g][i][e][o] 搜索 SEARCH_PATTERN，如果找到的话，就用后面的文本替换该模式。该函数将返回替换成功的数目。选项 g 表示跨行、全局的。选项 i 负责关闭大小写敏感。选项 e 将替换字符串作为一个表达式来求值，例如 s/\d+/$&+5/e
scalar	scalar(EXPR) 强制在标量型上下文语境中对 EXPR 求值
seek	seek(FILEHANDLE, POSITION, WHENCE) 在文件 FILEHANDLE 中的某个位置放置文件指针，该指针值是相对于其在文件中的位置 WHENCE 的。如果 WHENCE 值为 0，则表示从文件起始位置开始；如果 WHENCE 值为 1，则表示从文件当前位置开始；如果 WHENCE 值为 2，则表示从文件末尾开始。如果 WHENCE 值为 0，则 POSITION 的值不能为负数
seekdir	seekdir(DIRHANDLE, POSITION) 在与 DIRHANDLE 相关联的目录结构上为 readdir 函数设置 POSITION 参见 directory(3)
select	select(FILEHANDLE) select 如果省略 FILEHANDLE 的话，则返回当前选定的文件句柄；如果提供了 FILEHANDLE 的话，该函数将为 write 和 print 设置当前默认句柄 参见 formatting
select	select (RBITS, WBITS, EBITS, TIMEOUT) 检查 I/O 文件描述符，看它是否可读、可写或有其他没有确定的异常情况。该函数负责规定位掩码，其中 TIME 值以秒为单位 参见 select(2)
semctl	semctl(ID, SEMNUM, CMD, ARC) 调用 semctl 系统调用，以便实现对信号的控制操作。它具有特殊的返回代码，需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 semctl(2)

（续）

函　数	描　述
semget	semget (KEY, NSEMS, SIZE, FLAGS) 返回与 KEY 相关联的信号 ID。如果调用错误，则返回未定义值。需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 semget (2)
semop	semop (KEY, OPSTRING) 调用 semop 系统调用，对 KEY 标识的信号执行指定操作。OPSTRING 必须是一个包装好的 semop 结构数组。需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 semop (2)
send	send (SOCKET, MSG, FLAGS, TO) send (SOCKET, MSG, FLAGS) 把一条消息发送到套接字 SOCKET 上 参见 send(2)
setpgrp	setpgrp(PID, PGRP) 为指定的进程、进程组或用户设置进程组信息 参见 getpgrp(2)
setpriority	setpriority(WHICH, WHO, PRIORITY) 为指定的进程、进程组或用户设置当前优先权或 nice 值 参见 getpriority(2)
setsockopt	setsockopt(SOCKET, LEVEL, OPTNAME, OPTVAL) 对 SOCKET 设置请求的套接字选项 参见 getsockopt(2)
shift	shift(ARRAY) shift ARRAY shift 移出 ARRAY 的第一个值并返回它，同时让数组变短。如果省略了 ARRAY 的话，该函数就会移出 @ARGV 数组；如果是在子例程中的话，则移出 @_ 数组
shmctl	shmctl(ID, CMD, ARG) 调用 shmctl 系统调用，允许对共享内存执行控制操作。它具有特殊的返回代码。需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 shmctl (2)
shmget	shmget(KEY, SIZE, FLAGS) 返回与 KEY 相关联的共享内存段 ID，如果发生错误，则返回未定义的值。其所创建的共享内存段至少拥有 SIZE 个字节 本函数需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 shmget (2)
shmread	shmread(ID, VAR, POS, SIZE) 从共享内存段 ID 中的 POS 位置开始读取 SIZE 个字节内容。变量 VAR 用于保存读取得到的内容。该函数首先会附加段，然后读数据，最后分离段。本函数需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 shmmat (2)
shmwrite	shmwrite (ID, VAR, POS, SIZE) 从共享内存段 ID 中的 POS 位置开始写入 SIZE 个字节。变量 VAR 用于保存待写入的内容。该函数首先会附加段，然后写数据，最后分离段。本函数需要用到库文件 ipc.ph 和 sem.ph 参见 System V IPC 参见 shmmat (2)
shutdown	shutdown(SOCKET, HOW) 关闭 SOCKET 连接。如果 HOW 值为 0，则表示禁止进一步的接收操作；如果 HOW 值为 1，则表示禁止进一步的发送操作；如果 HOW 为 2，则禁止进一步的发送和接收操作 参见 shutdown(2)

（续）

函　数	描　述
sin	sin(EXPR) sin 返回 EXPR（以弧度表示）的正弦函数值。如果省略了 EXPR 的话，该函数则会返回 $_ 的 sin 值
sleep	sleep (EXPR) sleep EXPR sleep 使程序休眠 EXPR 秒。如果省略了 EXPR，程序将永远休眠 参见 sleep(3)
socket	socket(SOCKET, DOMAIN, TYPE, PROTOCOL) 打开指定类型的套接字，并将它附加到文件句柄 SOCKET 上 参见 socket(2)
socketpair	socketpair(SOCKET, SOCKET2 , DOMAIN, TYPE，PROTOCOL) 在指定域中创建属于指定类型的匿名连接套接字对 参见 socketpair(2)
sort	sort (SUBROUTINE LIST) sort (LIST) sort SUBROUTINE LIST sort LIST 对 LIST 进行排序，并返回排序后的数组。如果省略了 SUBROUTINE 的话，则按字符串顺序予以排序。如果规定了 SUBROUTINE 的话，则根据每个数组元素的排序方式给出返回大于、等于或小于 0 的子例程名称。两个待比较的元素将通过 $a 和 $b 变量传递给子例程，而不是通过 @_ 传递。SUBROUTINE 不能是递归函数 参见数组函数
splice	splice(ARRAY, OFFSET, LENGTH, LIST) splice(ARRAY, OFFSET, LENGTH) splice(ARRAY, OFFSET) 在数组中删除从 OFFSET 偏移量处开始，以 LENGTH 结束的元素。如果指定了 LIST，则以这个 LIST 替换删除的元素。该函数会返回从列表删除的元素。如果没有规定 LENGTH 的话，则删除从 OFFSET 到 ARRAY 末尾的所有元素
split	split(/PATTERN/, EXPR, LIMIT) split(/PATTERN/, EXPR) split(/PATTERN/) split 将 EXPR 拆分为含有字符串的数组，并将它返回到一个数组中。PATTERN 是用于分隔 EXPR 的定界符。如果省略了 PATTERN 的话，则使用空白字符作为定界符。LIMIT 负责规定拆分的最大分段数目
sprintf	$string=sprintf(FORMAT, LIST) 返回一个字符串，同时使用与 printf 函数相同的格式和规范把输出内容发送到 STDOUT 上 参见 printf(3)
sqrt	sqrt (EXPR) sqrt EXPR 返回 EXPR 的平方根。如果省略了 EXPR 的话，则返回 $_ 的平方根
srand	srand (EXPR) srand EXPR srand 为 rand 函数设置随机启动源。如果省略了 EXPR 的话，则默认会以 time 函数作为启动源

（续）

函　数	描　述
stat	stat(FILEHANDLE) stat FILEHANDLE stat(EXPR) 返回一个含有 13 个元素的数组，由 FILEHANDLE 或名为 EXPR 的文件状态构成。数组内含有如下内容： 设备 文件信息结点（inode）号 文件模式 指向文件的硬链接数目 拥有者的用户 ID 拥着者的组 ID 原始设备 文件大小 文件上一次访问时间 文件上一次修改时间 文件上一次状态变更时间 文件系统首选的 I/O 块大小 实际分配的块数 参见 stat(2)
study	study(SCALAR) study SCALAR study 使用链接列表机制，提高需要重复多次的模式匹配的搜索效率。如果省略了 SCALAR 的话，则默认使用 $_。该函数最适用于那些需要扫描很多常量字符串的循环
sub	sub NAME BLOCK sub NAME sub BLOCK sub NAME PROTO BLOCK sub NAME PROTO sub PROTO BLOCK 前两个声明负责指明子例程是否存在，不返回任何值。如果没有块的话，sub NAME 就是一个向前声明。sub BLOCK 用于创建匿名子例程。后三个声明与前面三个相似，但它们能通过原型来描述如何调用子例程。原型会提醒编译器，后面会出现子例程的定义，并告诉编译器该子例程期望的参数数目与类型。例如，sub foo($$@) 声明子例程 foo 需要三个参数，分别是两个标量型和一个数组型参数。如果传递的参数少于三个，则会发生错误
substr	substr(EXPR, OFFSET, LENGTH) substr(EXPR, OFFSET) 从 EXPR 中提取 OFFSET 处的子串，并返回得到的子串。如果规定了 LENGTH 的话，则从 OFFSET 开始获取长度为 LENGTH 的子串。如果 OFFSET 为负数的话，则表示从字符串的末尾开始提取
symlink	symlink(OLDFILE, NEWFILE) 创建符号链接。NEWFILE 将是 OLDFILE 的符号链接。这两个文件可以驻留在不同的分区上 参见 symlink(2)
syscall	syscall(LIST) syscall LIST 调用指定的 LIST 中第一个参数的系统调用。该列表中的系统调用都以 &SYS_ 开头，譬如系统调用 &SYS_system。LIST 中其余的项则是传递给该系统调用的参数。本函数需要用到 syscall.ph
sysopen	sysopen(FILEHANDLE, FILENAME, MODE) sysopen(FILEHANDLE, FILENAME, MODE, PERMS) 使用底层操作系统版本中的 open 调用打开文件句柄 FILEHANDLE。文件模式与具体用到的系统有关，可以在 Fcntl 库中找到。0 表示只读、1 表示只写、2 表示可读 / 写。如果省略了 PERMS 的话，则默认权限为 0666

（续）

函　数	描　述
sysread	sysread(FILEHANDLE, SCALAR, LENGTH, OFFSET) sysread(FILEHANDLE, SCALAR, LENGTH) 从文件句柄 FILEHANDLE 中读取 LENGTH 个字节到变量 SCALAR 中。它用到了 read 系统调用 参见 read(2)
sysseek	sysseek (FILEHANDLE, POSITION, WHENCE) 使用系统调用 lseek 设置文件句柄 FILEHANDLE 的系统位置，并绕过标准 I/O。如果 WHENCE 值为 0，则将新位置设置为 POSITION 处；如果是 1，则设置为当前位置加上 POSITION；如果是 2，则设置为 EOF 加上 POSITION（往往是一个负数）
system	system(LIST) system LIST 从 Perl 脚本中执行一段 shell 命令并返回其结果。该函数类似于 exec 函数，但它首先进行分支，并且让脚本等待命令执行完毕，然后才把控制权交回给脚本。其返回值是脚本的退出状态，可通过除以 256 或者右移低八位来获得 参见 system(3)
syswrite	syswrite (FILEHANDLE, SCALAR, LENGTH, OFFSET) syswrite (FILEHANDLE, SCALAR, LENGTH) syswrite (FILEHANDLE, SCALAR) 将变量 SCALAR 中 LENGTH 个字节写入文件句柄 FILEHANDLE 中，并返回写入句柄的字节数。如果规定了 OFFSET 的话，则从偏移量 OFFSET 处开始。它需要用到 write 系统调用 参见 write(2)
tell	tell(FILEHANDLE) tell FILEHANDLE tell 返回 FILEHANDLE 当前文件位置处的字节值（从字节 0 开始）。一般应当把返回值传递给 seek 函数，以便返回到文件内的某个位置上
telldir	telldir (DIRHANDLE) telldir DIRHANDLE 返回目录结构 DIRHANDLE 上 readdir 函数的当前位置 参见 directory(3)
tie	tie(VARIABLE, CLASSNAME, LIST) 把某个变量 VARIABLE 绑定到一个包（CLASSNAME）上，以便在实现其方法时可以使用该变量。LIST 由构造对象时传递给新方法的参数构成。该函数常用于关联数组，用于将它绑定到数据库上。其中的方法在包中具有预先定义的名称。在读取、存储或撤销绑定时，将自动调用这些预定义的方法 实现关联数组的包提供了下列方法： TIEHASH $classname, LIST DESTROY $self FETCH $self, $key STORE $self, $key DELETE $self, $key EXISTS $self, $key FIRSTKEY $self NEXTKEY $self, $lastkey 为数组提供了如下方法： TIEARRAY $classname, LIST DESTROY $self FETCH $self, $subscript STORE $self, $subscript, $value 为标量提供了如下方法： TIESCALAR $classname, LIST DESTROY $self FETCH $self STORE $self, $value 示例：

（续）

函　数	描　述
tie	`$object = tie %hash, Myhashclass` `while ($key, $value)=each (%hash){` `print "$key, $value\n" # invokes the FETCH method` `$object = tie @array, Myarrayclass` `$array[0]=5 # invokes the STORE method` `$object = tie $scalar, Myscalarclass` `untie $scalar # invokes the DESTROY method`
tied	tied VARIABLE 返回一个引用，指向之前使用 tie 函数连接的对象。如果 VARIABLE 尚未连接到包的话，则返回未定义值
time	time 返回一个 4 元素的数组，含有从 1970 年 1 月 1 日起算的 UTC 非闰秒时间。用于 gmtime 和 localtime 函数 参见 ctime(3)
times	times 返回一个 4 元素数组，负责以秒为单位告诉用户当前进程及其子进程的 CPU 时间 参见 times(3)
tr	tr/SEARCHPATTERN/REPLACEMENT/[c][d][e] y/SEARCHPATTERN/REPLACEMENT/[c][d][e] 将 SEARCHPATTERN 中的字符替换为 REPLACEMENT 中的对应字符。类似于 UNIX 中的 tr 命令
truncate	truncate (FILEHANDLE, LENGTH) truncate (EXPR, LENGTH) 将 FILEHANDLE 或 EXPR 的内容截短为指定的长度 参见 truncate(2)
uc	uc EXPR 返回 EXPR 的大写形式（如果没有提供 EXPR 的话，则返回 $_ 的大写形式）。该函数与 \U\E 转义序列功能相同
ucfirst	ucfirst EXPR 返回 EXPR 首字母大写后的结果（如果没有提供 EXPR 的话，则处理 $_）。等效于 \u 转义序列
umask	umask (EXPR) umask EXPR umask 为进程设置 umask（或为文件创建掩码），并返回旧的 umask。当省略了 EXPR 时，该函数将只返回当前的 umask 掩码值 参见 umask(2)
undef	undef (EXPR) undef EXPR undef EXPR 未定义，即 lvalue。可用于标量、数组、散列或子例程名（&subroutine），以便恢复与之关联的存储空间。该函数始终返回未定义值。当在子例程中使用该函数返回时，亦可用于判断是否发生了错误
unlink	unlink (LIST) unlink LIST unlink 删除 LIST 中列举的文件，并返回删除的文件数目。如果没有提供参数的话，则断开 $_ 变量值的连接 参见 unlink(2)
unpack	unpack(TEMPLATE, EXPR) 对代表结构的字符串进行解压，将它展开为数组，并返回数组值。该函数使用 TEMPLATE 来获得值的顺序和类型。该函数与 pack 函数功能相反 参见 pack
unshift	unshift (LIST) unshift 将 LIST 添加到数组的开头位置，并返回新数组中的元素数目
untie	untie VARIABLE 断开变量与包之间的连接，其功能与 tie 相反

（续）

函　数	描　述
use	use MODULE VERSION LIST use MODULE LIST use MODULE use MODULE() use pragma 这是一个编译器指令，负责从 MODULE 中将其子例程和变量导入到当前的包。VERSION 是 Perl 的当前版本号。LIST 含有当前包需要导入的变量和子例程的实际名称。如果不想向命名空间中导入任何东西，则可使用空参数。-m 和 -M 标记可以在命令行中代替 use。编译指示是一种特殊的模块，可以在编译时影响语句的行为。常用的三个编译指示包括：integer、subs 和 strict
utime	utime (LIST) utime LIST 更改列表中各个文件的访问时间和修改时间。LIST 的前两个元素是数值型的访问时间和修改时间
values	values(%ASSOC_ARRAY) values ASSOC_ARRAY 返回一个数组，该数组由关联数组 ASSOC_ARRAY 中所有的值以随机顺序排列而成
vec	vec (EXPR , OFFSET, BITS) 将字符串 EXPR 当作无符号整数向量进行处理。该函数将返回指定元素的值。OFFSET 是为了找到所需元素而应当跳过的元素数目。BITS 是向量中每个元素的比特数。BITS 必须是介于 1 到 32 之间的 2 的整数幂，即 1、2、4、8、16 或 32
wait	wait 等待子例程执行结束。该函数会返回结束进程的 pid；如果没有子例程的话，则返回 -1。其退出状态值将返回到变量 $? 中 参见 wait(2)
waitpid	waitpid(PID, FLAGS) 等待子例程执行结束，并返回进程结束时间。如果没有子例程，或者 FLAGS 规定为非阻塞而进程已经结束的话，该函数就会返回 -1。其退出状态值将返回到变量 $? 中。该函数需要用到 sys/wait.ph 参见 wait(2)
wantarray	wantarray 如果当前运行的子例程需要返回数组值的话，返回真；也就是要把子例程的返回值赋值给数组。如果要得到的是标量值的话，返回假。例如：return wantarray? (): undef
warn	warn (LIST) warn LIST 发送警告消息到 STDERR，类似于 die 函数，但它不退出程序
write	write (FILEHANDLE) write FILEHANDLE write 将格式化后的记录写入到文件句柄 FILEHANDLE 或当前选定的 FILEHANDLE 中；即不加参数地为 FILEHANDLE 调用相应的格式（形象行）。其输出内容可以写入 STDOUT，亦可写入由 select 函数选中的当前 FILEHANDLE 中。该函数与系统调用 write(2) 无关 参见 syswrite
y	y/SEARCHPATTERN/REPLACEMENT/[c][d][e] 将 SEARCHPATTERN 中的字符替换为 REPLACEMENT 中的对应字符。也称作 tr，类似于 UNIX 中的 tr 命令或 sed y 命令

A.2　特殊变量

表 A-2　文件句柄

变　量	用　途
$|	如果非 0，则在对当前选定的文件执行写或打印操作后强制清除缓冲区

（续）

变　量	用　途
$%	当前选中文件句柄的当前页码
$=	当前选中文件句柄的当前页面长度
$-	当前选中文件句柄的页面剩余长度
$~	当前选中文件句柄的报表格式名称
$^	当前选中文件句柄的当前页眉格式名称

表 A-3　块内局部变量

变　量	用　途
$1..$9	含有与应用对应的一组括号的子记忆模式。（等效于 \1..\9）
$&	上一次与模式相匹配的字符串（等效于 sed 编辑器）
$`	上一次与模式相匹配的项的前导字符串
$'	上一次与模式相匹配的项之后的字符串
$+	与搜索模式相匹配的最后一个模式

示例 A.1

```
$str="old and restless";

print "$&\n" if $str =~ /and/;
print "$'\n" if $str =~ /and/;
print "$'\n" if $str =~ /and/;
print "\nold string is: $str\n";
$str=~s/(old) and (restless)/$2 and $1/;
print "new string is: $str\n";
print "\nlast pattern matched: $+\n";

(Output)
and
old
restless
old string is: old and restless
new string is: restless and old
last pattern matched is: restless
```

表 A-4　全局变量

变　量	用　途
$_	默认的输入和模式搜索空间
$.	读取上一个文件句柄的当前输入行号；必须在关闭前一个文件句柄后才能对下一个文件句柄重置行号
$/	输入记录的分隔符，默认为换行符（类似于 awk 中的 RS）
$\	打印函数的输入记录分隔符。除非设置了 $\="\n"，否则不会打印换行符
$,	打印函数的输出字段分隔符。一般在以逗号隔开的字符串中不打印其定界符，除非设置了 $,=" "

（续）

变　量	用　途
$"	与 $, 类似，但它是在双引号中提供打印数组的，默认值为空格
$#	使用打印函数打印的数字输出格式（类似于 awk 中的 OMFT）
$$	运行本脚本的 Perl 程序进程 ID
$?	上一个关闭的管道、反引号中的命令以及系统函数的返回状态
$*	默认值为 0。如果为 1 的话，表示在字符串中进行多行匹配；0 表示只在单行中匹配
$0	Perl 脚本名
$[	指向数组或子串的第一个字符的索引，默认值为 0
$]	当使用 Perl−v 命令获得版本信息时打印出字符串的第一部分
$;	多维数组的下标分隔符。默认为 \034（类似于 awk 的 SUBSEP）
$!	如果是数字，产生 errno（系统错误号）的当前值，以及与之对应的系统错误字符串
$@	上一个 eval、do 或 require 函数产生的错误消息
$<	进程的实际 uid
$>	进程的有效 uid
$(	进程的实际 gid
$)	进程的有效 gid
$:	字符集，将其后的字符串断开，并以某种格式填入连续的行（从 ^ 处开始）。默认是以空白字符、换行符或冒号上隔开的 \n-
$^A	formline 和 write 操作的累加器
$^C	如果 Perl 使用 −c 选项在只编译的模式下运行，则返回 TRUE
$^D	使用 −D 开关打开 Perl 调试标记
$^E	操作系统的相关出错信息
$^F	传递给子例程的最大文件描述符数，一般是 2
$^H	语法检查的当前状态
$^I	在使用 −i 开关时，原位编辑扩展的当前值
$^L	进纸符，用在格式中
$^M	紧急内存池
$^O	操作系统名
$^P	内部 Perl 调试标记
$^S	Perl 解释器的状态
$^T	脚本执行时间。用于 −A、−C 和 −M 测试运算符，可设置为 time 返回的任何数字值，执行与当前时间有关的文件测试
$^V	Perl 版本
$^W	警告开关的当前设定情况

（续）

变　量	用　途
$^X	Perl 调用的全路径名
_	下划线，负责在声明文件时测试文件的特殊标志符
ARGV	用于行参数循环的特殊文件句柄数组
$ARGV	从 <ARGV> 读取时，含有当前文件名的变量
@ARGV	含有命令行参数的数组
DATA	一个特殊文件句柄变量，负责引用 _END_ 标志后面的内容
@F	在使用 −a 开关时，自动拆分输入行所得到的数组
@INC	一个含有文件名路径的数组，require 和 do 函数将从中搜索将要导入的文件名路径
%INC	一个含有文件名路径的关联数组，require 和 do 函数将从中搜索将要导入的文件名路径。其中键是文件名，值是文件路径位置
%ENV	含有当前环境变量的关联数组
@EXPORT	导出时的默认符号
@EXPORT_OK	根据用户请求而导出的符号
%EXPORT_TAGS	用于 Exporter.pm 中，负责对符号集合统一命名
%SIG	用于设置信号处理例程的关联数组
STDERR	用于标准出错内容的特殊文件句柄
STDIN	用于标准输入的特殊文件句柄
STDOUT	用于标准输出内容的特殊文件句柄

A.3　Perl 编译指示

编译指示是一种特殊的“伪”模块，负责规定编译器的行为。use declaration 用于将编译器指定（又称编译指示）导入到 Perl 程序中。编译指示规定了如何编译语句块；它们是针对词法的，其作用域仅限于所在的代码块，用户可通过 no 指定予以关闭。编译指示的名称一般都是小写的。表 A-5 列出了部分 Perl 编译指示。

表 A-5　Perl 编译指示

编译指示	用　途
use autouse	提供了一种机制，使得只有调用模块中的函数才能在运行时加载该模块
use base	让程序员根据编译时列出的父类声明派生类，从而不再需要使用 require。例如，use base qw(A B); 等效于 BEGIN{ require A; requireB::push(@ISA, qw(A B));}
use bytes	在 Perl 5.6 之前，所有字符串都当作一系列字节来处理。现在，字符串可以含有比数字编码字节范围更广的字节。bytes 编译指示用于规定代码使用上述旧的面向字节的语义
use constant	声明一个命名符号，代表指定的标量或列表常量。例如： use constant BUFFER_SIZE=>4096; use constant OS=> 'Solaris';

（续）

编译指示	用　途
use diagnostics	在 Perl 编译器和解释器发出的诊断信息之外，强制显示更详细的信息。由于它只能影响到最内层的块，因此一般需要把编译指示放在程序开头位置。不能使用 no diagnostics
use integer	这是一个词法域的编译指示，告诉编译器以整数算术处理所有的算术操作，即在执行这些操作时舍去浮点数的小数部分
use locale	这是一个词法域的编译指示，告诉编译器在处理正则表达式、内建函数或字符转换等操作时启用或关闭 POSIX 现场
use open	声明一个或多个 I/O 操作的规范；当前支持的两种规范是 :raw 和 :ctlf
use overload	负责在使用对象时重新定义内建操作的涵义。有关重载运算符的示例请读者参见标准 Perl 库中的 Math::BigFloat
use strict 'vars'	使用 'vars' 参数，要求必须使用词法变量（my）或带有包名和作用域运算符的完全限定的变量名与导出变量，否则就会编译出错
use strict 'ref'	如果脚本中用到了符号引用的话，则产生运行时错误，譬如 typeglob
use strict 'subs'	如果用到了裸字（bareword）并且它不是预先定义的子例程或文件句柄的话，就触发编译时错误
use strict	如果用到了符号引用、声明了非词法变量或者用到了非子例程或文件句柄的裸字的话，则产生编译时错误
use vars qw(list)	在引入 our 之前用于声明全局变量
use warnings	这是一个词法域的编译指示，告诉编译器更灵活地控制 Perl 的内建警告，譬如 −w 开关或者 $^W 变量
use lib 'library path'	在编译时载入库，而不是在运行时载入
use sigtrap 'signal names'	指定一组符号处理程序，用于初始化列出的符号。如果不使用一组默认符号作为参数的话，则会打印程序堆栈信息，并发出 ABRT 信号
use subs qw (subroutine list)	预先声明的子例程列表，使得可以不用括号来调用列出的子例程，并覆盖原有的内建函数
no integer	如需关闭或不输出编译指示，只需在该编译指示前面加上 no 即可

A.4　Perl 模块

表 A-6　编程通用模块

模　块	描　述
Benchmark	检查并比较运行代码的 CPU 时间速度
Config	访问 %Config 散列的 Perl 配置选项
Env	将 %ENV 散列转换为含有环境变量的标量，譬如把 $ENV{HOME} 转换为标量 $HOME
English	提供特殊变量的 English 或 awk 名称。例如，$0 可用 $PROGRAM_NAME 来命名
Getopt	使用参数处理命令行选项和开关
Shell	在 Perl 脚本中负责以子例程的形式执行 shell 命令。例如，$today=date();
Symbol	使用 gensym() 产生匿名 glob，并使用 qualify() 来限定变量名

表 A-7　CGI

模　块	描　述
CGI	CGI（通用网关接口）类
CGI::Apache	用于 CGI.pm 和 Perl-Apache API
CGI::Carp	用于处理 HTTP 出错消息，并负责产生错误日志文件
CGI::Cookie	与 Netscape cookies 进行交互
CGI::Fast	与 Fast Perl 接口进行交互
CGI::Pretty	产生 Pretty 格式的 HTML 代码
CGI::Push	到服务器 Push 的简单接口

表 A-8　错误处理

模　块	描　述
Carp	产生类似于 die 的错误消息，报告发生错误的调用例程的行号。可从该模块中调用的子例程包括 carp()、croak() 和 confess()
Errno	加载 libc 的 errno.h 定义
Sys::Syslog	提供与 UNIX syslog(3) 交互的 Perl 接口

表 A-9　文件处理

模　块	描　述
Cwd	获得当前工作目录的路径名。如用于 −w 开关，则会产生错误消息
DirHandle	为目录句柄提供面向对象的接口
Fcntl	加载 libc 的 fcntl.h（文件控制）定义
File::Basename	将文件名拆分为多个组成部分，或者从全目录路径中提取文件名或目录名
File::CheckTree	对目录树中的某个文件集合执行文件测试操作
File::Copy	用于拷贝文件或文件句柄
File::DosGlob	进行类似于 DOS 的 glob 操作
File::Find	用于在 UNIX 文件树中查找
File::Finddepth	深度优先搜索文件系统
File::Glob	对 UNIX 文件名进行 glob 操作
File::Path	创建或删除一系列目录
File::Spec	对文件名进行移植操作
FileCache	允许打开比系统允许的数量更多的文件
FileHandle	为文件句柄访问方法提供面向对象的接口
SelectServer	保存和恢复选择的文件句柄
flush.pl	写入文件句柄缓冲区中剩余的所有数据，或者在打印表达式后清除缓冲区
pwd.pl	使用 chdir 后将 PWD 环境变量设置为当前的工作目录
stat.pl	将 stat 函数返回的值赋值给标量：$st_dev、$st_ino、$st_mode、$st_nlink、$st_uid、$st_rdev、$st_atime、$st_mtime、$st_ctime、$st_blksize、$st_blocks

表 A-10　文本处理

模　块	描　述
Pod::Text	将 Pod 文档转换为 ASCII 格式的文档
Search::Dict	在目录文件中搜索字符串（按照字母表顺序），并将文件指针设置到下一行
Term::Complete	为了提示用户，可通过按下 Tab 键完成其余部分的输入，或通过按下 <Ctrl>−d 完成列表的选择，以便提供类似于文件名补齐功能的接口
Text::Abbrev	创建缩略表，它是由列表中的键 / 值对构成的散列。其中的键是缩写内容，值是原字符串；譬如 ma/mail、mo/more
Text::ParseWords	像 shell 一样把一行文本解析为一系列词语，同时忽略开头的空白
Text::Soundex	将一个字映射为长度为四个字符的与该字发音完全对应的代码
Text::Tabs	将制表符扩展为空格，或者不把制表符扩展为空格
Text::Warp	将文本换行为独立的段落

表 A-11　数据库接口

模　块	描　述
AnyDBM_File	一个基于 UNIX 系统的模块，提供了多种 DBM 架构
DB_File	提供对 Berkeley DB 管理系统的访问手段。参见 ftp://ftp.cs.berkeley.edu/ucb/4bsd
DBI	返回系统上的数据库列表及其驱动程序，用于与数据库交互
GDBM_File	提供对 GUN 数据库管理系统的访问手段。参见 ftp://prep.ai.mit.edu/pub/gnu
NDBM_File	一个基于 UNIX 系统的模块，提供了与 NDBM 文件交互的手段
ODBM_File	一个基于 UNIX 系统的模块，提供了与 ODBM 文件交互的手段
SDBM_File	一个基于 UNIX 系统的模块，提供了与 SDBM 文件交互的手段

表 A-12　数学

模　块	描　述
bigrat.pl	对分数启用无限精度运算
Math::BigFloat	支持任意大小的浮点运算
Math::BigInt	支持任意大小的整数运算
Math::Complex	支持复杂的数字，并展示运算符重载
Math::Trig	支持三角函数

表 A-13　网络

模　块	描　述
chat2.pl	允许 Perl 操纵交互式的网络服务，如 FTP
comm.pl	比 chat2.pl 版本更新。允许 Perl 操纵交互式的网络服务
IPC::Open2	打开进程进行读写，使得能够与外部程序相互传输数据

（续）

模　块	描　述
IPC::Open3	打开进程进行读写或处理错误信息，使得能够与外部程序相互传输数据
Net::Ping	检查远程主机是否开启
Socket	创建套接字，为进程间通信导入套接字方法，并加载 socket.h 头文件
Sys::Hostname	获得系统的主机名

表 A-14　时间和位置

模　块	描　述
I18N::Collate	根据当前时区比较 8 比特的标量数据
Time::gmtime	使用 Perl 内建函数 gmtime() 的接口
Time::Local	根据本地 GMT（UTC）时间计算 UNIX 时间（从 1970 年 1 月 1 日开始非闰秒数）
Time::localtime	使用 Perl 内建函数 localtime 的接口

表 A-15　终端

模　块	描　述
Term::Cap	提供操作终端配置的底层功能，指向 termcap 数据库的终端接口

表 A-16　面向对象的模块函数

模　块	描　述
Autoloader	对于大模块，将只加载需要的部分
AutoSplit	为了支持自动加载，将模块拆分为指定字节大小的程序块
Devel::SelfStubber	为自加载模块生成代码存根，以便在调用方法时能顺利加载
DynaLoader	用于自动和动态加载模块
Exporter	供其他模块使用，负责导入方法和变量，以便使之可用
overload	重载算术操作
Tie::Hash	提供将散列与包相连接的方法
Tie::Scalar	提供将标量与包相连接的方法
Tie::SubstrHash	为含有常量键和记录大小的数组提供类似于散列的接口

表 A-17　语言扩展

模　块	描　述
ExtUtils::Install	用于安装和卸载与平台相关的 Perl 扩展
ExtUtils::Liblist	决定使用的库，以及如何使用它们

（续）

模　块	描　述
ExtUtils::MakeMaker	在扩展库中为 Perl 扩展创建 Makefile
ExtUtils::Manifest	自动维护由文件名列表构成的 MANIFEST 文件
ExtUtils::Miniperl	编写含有引导程序文件的 C 代码 perlmain.c，用于创建供 Perl 模块使用的存档库
ExtUtils::Mkbootstrap	从扩展的 Makefile 中调用，负责创建引导程序文件，以便动态加载某些系统
ExtUtils::Mksysmlists	编写在创建动态扩展的共享库时所要用到的一些链接的 linker 选项文件
ExtUtils::MM_OS2	覆盖方法的实现，并影响 UNIX 行为
ExtUtils::MM_UNIX	用于 MakeMaker，负责为 UNIX 与非 UNIX 系统提供方法
ExtUtils::MM_VMS	覆盖方法的实现，并影响 UNIX 行为
Fcntl	翻译 C 的 fcntl 头文件
POSIX	为 IEEE std 1003.1 标识符提供 Perl 处理接口
Safe	提供私有分隔区域，用于评估不安全的 Perl 代码
Test::Harness	用于 MakeMaker，负责测试运行 Perl 扩展，并生成诊断信息

A.5　命令行开关

表 A-18　命令行开关

开关项	描　述
-0	规定记录分隔符
-a	与 -n 或者 -p 一起使用，负责打开自动拆分模式，用于对空白字符进行隐式拆分，并将字段放入数组 @F 中，例如： date | Perl -ane 'print "$F[0]\n";
-c	只检查 Perl 脚本语法，而不执行脚本
-d	对脚本打开 Perl 调试器
-D	设置 Perl 的调试标记（请先检查 Perl 的安装情况，确保已经安装了调试器）。若要观察 Perl 是如何执行脚本的，可使用 -D14
-e command	用于在命令行而不是在脚本中执行 Perl 命令
-F pattern	规定拆分输入行时使用的模式。模式是位于斜杠、单引号或双引号之间的正则表达式。例如，-F/:+/ 表示在一个或多个冒号上拆分输入行。如果 -a 仍然起作用的话，则打开它
-h	打印 Perl 的命令行选项列表
-i extension	在使用 <> 循环遍历文件时启用原位编辑模式。如果没有规定扩展名的话，则原位修改各行内容。否则使用扩展名来修改输入文件名（以便充当备份文件），并使用原位编辑的原文件名创建输出文件。这也是所有 print 语句选择的文件句柄
-I directory	与 -P 选项一起使用，负责告诉 C 预处理程序查找包含的文件，其默认目录包括 /usr/included、/usr/lib/Perl 以及当前目录
-l digits	启用自动行结束处理。如果使用了 -n 或 -p 选项，则去掉行终止符。把位数（八进制）赋值给 $\，并将行终止符添加到 print 语句中。如果没有规定位数的话，则将 $\ 设置为 $/ 的当前值（参见表 A-2“特殊变量”）

（续）

开关项	描　述
-m [-] module -M [-] module -M [-] 'module' -[mM] module = arg [,arg]..	
-mmodule	在执行 Perl 脚本之前执行 use 方法
-Mmodule	在执行 Perl 脚本之前执行 use 方法。如果添加附加的文本，则可使用引号。方括号中的短横线表示把 use 指令替换为 no
-n	使 Perl 隐式地循环遍历指定的文件，并只打印规定的行
-p	使 Perl 隐式地循环遍历指定的文件，同时打印所有的行
-P	使要运行的脚本在 Perl 编译之前通过 C 预处理程序
-s	在脚本名之后、文件名参数之前启用开关项解析，并从 @ARGV 数组中删除所有找到的开关项。将开关项设置为同名的标量型变量，并将标量赋值为 1。例如，-abc 在脚本中将转换为 $abc
-S	如果不支持 #!/usr/bin/Perl 行，则让 Perl 在 PATH 环境变量中搜索脚本
-T	强制打开"污染"检查，用于测试脚本，一般只用在 setuid 或 setgid 程序上。推荐读者自行检查 CGI 脚本
-u	编译后产生脚本的核心转储（基于 UNIX 系统）
-U	允许 Perl 执行不安全的操作，如果是超级用户的话，则断开目录链接
-v	打印 Perl 的版本信息
-V	打印最重要的 Perl 配置项和数组 @INC 中当前值的汇总
-V:NAME	打印 NAME 的值，其中 NAME 是配置变量
-w	打印警告信息，包括错误使用保留字、文件句柄、子例程等情况
-W	启用所有警告，而不论是否已经用 no warnings 关闭了本地警告
-xdirectory	忽略 #!/usr/bin/Perl 行之前的任何文本。如果将目录名当作 -x 开关的参数，则 Perl 会在开始执行脚本之前自动切换到该目录
-X	关闭所有警告

A.6　调试器

A.6.1　获取有关调试器的信息

若要获得有关如何使用调试器的信息，可在命令行中键入如下命令

```
perldoc perldebug
```

下面是一个输出的示例：

```
NAME
  perldebug - Perl debugging

DESCRIPTION
```

```
First of all, have you tried using the -w switch?
```

A.6.2 Perl 调试器

如果在调用 Perl 时使用了 -d 开关项，则用户脚本就会执行在 Perl 源代码调试器中。它类似于一个交互式的 Perl 环境，能够提示用户通过一些调试命令来检查源代码、设置断点、获取堆栈跟踪信息、修改变量值等。由于调试器是如此方便，很多用户干脆直接启动它，从而能交互地检查 Perl 程序的工作流程。例如：

```
$ Perl -d -e 42
```

在 Perl 中，通常情况下调试器都不是一个独立的程序（尽管它看上去很像）。读者可使用 -d 标志要求编译器把源码信息插入到将要传递给解释器的解析树中。这就是说，在调试器能起作用前，这段代码必须成功通过编译。然后，解释器会在启动时自动加载一个含有调试器的特殊 Perl 库文件。

程序将在第一个运行时可执行语句前面挂起（但位于编译时语句后面），并要求用户输入一个调试命令。与人们猜想的不同，每当调试器挂起并显示一行代码时，显示出来的总是下一行将要执行的代码，而不是刚刚执行完毕的代码。

调试器不能识别的任何命令都将直接当作当前包中的 Perl 代码来执行（处理）。（调试器使用 DB 包来维护自己的状态信息。）

对于任何在调试器提示符下输入的文本内容，在进一步处理之前都会去除其首尾的空格，如果某个调试器命令与读者程序中的某个函数名发生冲突的话，用户只需在函数名前面加上一些字符，以使它看上去不像一个调试器命令，譬如加上一个 ; 号、一个 + 号、或者用大括号或小括号把它围起来。

<continues here>

A.6.3 进入和退出调试器

如需调用 Perl 调试器，可使用 -d 开关项。它用于在成功编译后启动一种交互式类型的环境检查程序。在每一行执行结束后，脚本都会挂起，并要求用户提供命令。用户看到的行是将要执行的下一行脚本，而不是刚刚执行完毕的前一行。其提示符含有当前的包、函数、文件、行号和当前行信息。下面是所有调试器命令的列表。

在启动调试器后，读者还可通过在调试器提示符下键入 h 命令来列出所有的调试器命令；如果无法阅读，也可键入 h h 命令。

如需退出调试器，可键入 q 表示退出，或者键入 R 表示重新启动。

```
$ perl -d exer.1

Loading DB routines from $RCSfile: perldb.pl,v $$Revision: 4.0.1.2
$$Date: 91/11/05 17:55:58 $
Emacs support available.
Enter h for help.
main'(exer.1:3):        print "Today is ", 'date';
  DB<1> h
T                      Stack trace.
```

```
s                       Single step.
n                       Next, steps over subroutine calls.
r                       Return from current subroutine.
c [line]                Continue; optionally inserts a one-time-only
                        breakpoint at the specified line.
<CR>                    Repeat last n or s.
l min+incr              List incr+1 lines starting at min.
l min-max               List lines.
l line                  List line.
l                       List next window.
-                       List previous window.
w line                  List window around line.
l subname               List subroutine.
f filename              Switch to filename.
/pattern/               Search forwards for pattern; final / is optional.
?pattern?               Search backwards for pattern.
L                       List breakpoints and actions.
S                       List subroutine names.
t                       Toggle trace mode.
b [line] [condition]    Set breakpoint; line defaults to the current
                        execution line; condition breaks if it evaluates
                        to true, defaults to 1.
b subname [condition]   Set breakpoint at first line of subroutine.
d [line]                Delete breakpoint.
D                       Delete all breakpoints.
a [line] command        Set an action to be done before the line is
                        executed. Sequence is: check for breakpoint, print
                        line if necessary, do action, prompt user if
                        breakpoint or step, evaluate line.
A                       Delete all actions.
V [pkg [vars]]          List some (default all) variables in package
                        (default current).
X [vars]                Same as "V currentpackage [vars]".
< command               Define command before prompt.
> command               Define command after prompt.
! number                Redo command (default previous command).
! -number               Redo numberth-to-last command.
H -number               Display last number commands (default all).
q or ^D                 Quit.
p expr                  Same as "print DB'OUT expr" in current package.
= [alias value]         Define a command alias, or list current aliases.
command                 Execute as a Perl statement in current package.
   DB<1> l
3:      print "Today is ", 'date';
4:      print "The name of this \uperl script\e is $0.\n";
5:      print "Hello. The number we will examine is 125.5.\n";
6:      printf "The \unumber\e  is %d.\n", 125.5;
7:      printf "The \unumber\e  is %d.\n", 125.5;
8:      printf "The following number is taking up 20 spaces and is
        right-justified.\n";
9:      printf "|%-20s|\n", 125;
10:     printf "\t\tThe number in hex is %x\n", 125.5;
11:     printf "\t\tThe number in octal is %o\n", 125.5;
12:     printf "The number in scientific notation is %e\n", 125.5;
  DB<1> q     (quit)
```

A.6.4 调试器命令

获取帮助：

h	列出所有调试器命令的帮助信息
h p	列出调试器命令 p 的帮助信息

列出部分脚本：

l	列出程序的 10 行内容
l 8	列出第 8 行
l 5-10	列出第 5 行到第 10 行脚本
l greetme	列出子例程 greetme 中的脚本
L	列出待执行的下一行脚本
w7	列出一个含有指定的第 7 行脚本内容的窗口。列出指定行之前的 3 行脚本，并将其后各行填入该窗口
/^abc/	向前搜索正则表达式 abc，其中 abc 位于行首
?abc?	向后搜索正则表达式 abc
S	遵照包名、双冒号和子例程名的格式，列出程序中所有的子例程
r	执行当前子例程中其余的语句，然后显示紧接着该子例程调用的行

逐行递进：

s	一次一行地递进遍历脚本
n	类似于 s，但它会执行子例程调用，而不是逐行经过它们
Enter	按下 Enter 健，重复前一个 s 或 n 命令
.	重复执行上一行脚本
_	重复执行当前行前面的所有脚本行
r	继续执行，直到当前执行的子例程返回；并在返回后显示其返回值

退出调试器：

q	退出调试器
<Ctrl>-d	退出调试器
R	重新启动调试器和新会话

断点：断点负责设定程序暂停的位置，以便对它进行检查。必须将它设置在可执行语句的行首。

b45	在第 45 行设置断点。程序将在第 45 处挂起，键入 c 则继续执行
c	继续执行
b greetme	在子例程 greetme 上设置断点
b $x>10	当条件为真时触发断点

w	在断点周围创建一个窗口，并标记找到断点的行；例如 10==>b（断点位于第 10 行）
d	删除位于待执行的行上面的断点
d12	删除第 12 行上的断点
D	删除所有断点

打印变量值：

X name	显示变量 name 的值。在变量前不需要加上表示其类型的字符；例如，应当使用 x，而不是 $x 或 @x
V package	显示包中所有的变量
P $x+3	求值，并打印表达式

跟踪：

T	产生子例程调用内容的回溯清单
t	切换跟踪模式

别名：

= 列出所有别名

= ph print "$hashref->{Science}->{Lou}"	ph 是用于打印散列值的别名

附录 B　SQL 语言入门

B.1　什么是 SQL

当读者访问 Google 并请求信息时，这种请求又称作查询（query）。搜索引擎将会收集匹配于该查询的任何 Web 页面。为了缩小搜索范围，读者可以借助更多的描述性关键字来提炼请求。数据库检索过程也与之相似。当向数据库发出请求时，该请求应当遵循某种特定的格式，数据库服务器将会尝试定位所需的信息，并返回一个结果。读者使用的查询语言决定了数据库的查询方式。与关系型数据库进行通信的标准语言是 SQL，即结构化查询语言（Structured Query Language）。SQL 是一种符合 ANSI（美国国家标准署）标准的计算机语言，它设计得尽量与英语靠拢，因此是一种非常易于学习的语言。当前流行的数据库管理系统如 Oracle、Sybase 和 Microsoft SQL Server 都使用 SQL。尽管有些系统针对该语言创建了自己的专用扩展，但其标准的基本命令（譬如 SELECT、INSERT、DELETE、UPDATE、CREATE 和 DROP）就足以帮助用户完成在处理数据库操作时碰到的绝大多数任务了。

SQL 语言的历史可以追溯到 E. F“Ted”Cobb，IBM 公司的一位研究人员，他在 1970 年 6 月首先发表的一篇论文为关系型数据库理论奠定了基础。它是一种类似于英语的计算机语言，用于和关系型数据库进行通信。Cobb 的论文促使 IBM 启动了一项大型研究项目，设计了一套名叫 System/R 的关系型数据库系统和一种名为 SEQUEL（结构化英语查询语言）的数据库语言，后者就是今天的 SQL（发音为“see-quell”）。在 20 世纪 70 年代晚期，另外两家公司也跟进开发了与之相似的产品，后来分别发展成为 Oracle 和 Ingres。到了 1985 年，Oracle 号称已经安装了超过 1,000 份；而到 20 时间 90 年代早期，SQL 已经成为在大中型机构中实现数据库管理的标准，尤其在 UNIX 和巨型计算机上更是如此。

B.1.1　标准化 SQL

和英语中的方言类似，SQL 也有多种不同的风格。目前使用的 SQL 是基于 IBM 最初实现而开发的，并增添了许多附加功能。为帮助大家明确语言应当支持哪些特性，标准就应运而生了。在 1986 年，ANSI 发布了 SQL 标准，并在 1989 年、1992 年以及 1999 年分别对它进行了修订。目前最为常用的标准是 SQL92，即对最初规范（SQL2）的第二次修订。大多数商用数据库（MySQL、Oracle、Sybase、Microsoft Access 和 Microsoft SQL Server）都完全支持 SQL，并声称与标准百分之百兼容。不过，标准本身就非常复杂，再加上与英语中的方言一样，各个开发商也在它们的 SQL 版本中添加了各种扩展组件，因此很难保证某个应用程序能够运行在所有的 SQL 服务器数据库上。

在本附录中，我们将专注于基本的 SQL 语言，并学习诸如创建表、插入、删除和数据选择之类的概念。

B.1.2 执行 SQL 语句

由于本书讨论的数据库管理系统是 MySQL，因此下面的示例都将使用 MySQL 数据库服务器;尽管读者可能更喜欢 MySQL 查询浏览器（MySQL Query Browser），但这里我们将在 mysql 命令行客户端中执行大部分的 SQL 命令。在连接到数据库之后，读者只需按照第 17 章介绍的那样在 mysql 控制台（即命令行窗口，见图 B-1）中键入命令。

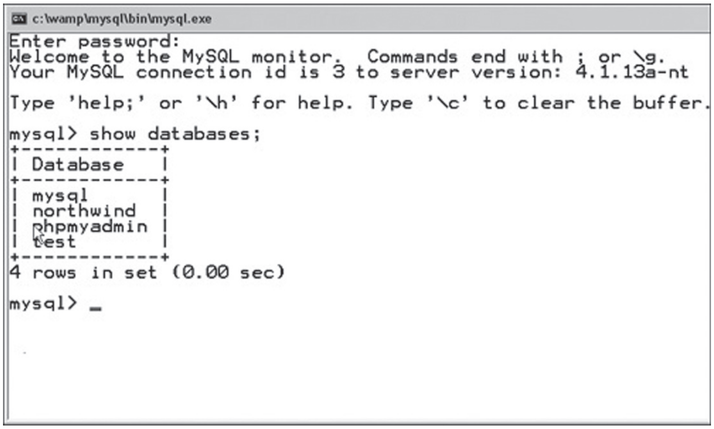


图 B-1　mysql 控制台

MySQL 查询浏览器。如需在 MySQL 查询浏览器中运行 SQL 命令，读者可以在应用程序窗口顶部的输入框中键入这些命令，然后单击执行（Execute）按钮。

当按下执行按钮（即位于查询窗口右侧的绿色按钮）后，结果将会以 Resultset 标签页（Resultset tab）的形式显示在应用程序的中央（见图 B-2）。

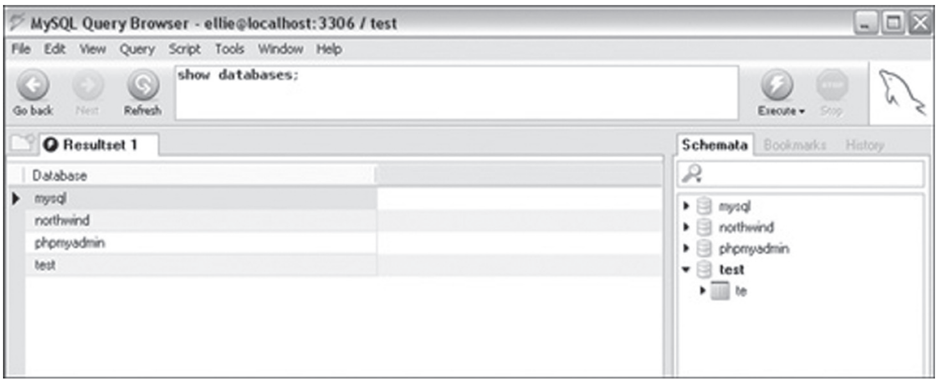


图 B-2　MySQL 查询浏览器 GUI

B.1.3 关于 SQL 命令 / 查询

SQL 是一种计算机语言，和其他语言一样，SQL 也拥有它自己的规则、语法以及一组特殊字或保留字。多年来，该语言出现了很多不同的变种，因为各个关系型数据库开发商在竞争的名义下向它添加了许多额外的数据处理特性。本节只涉及基本的 SQL 命令和标记。

由于 SQL 拥有很多命令，因此把这些命令分成了两大类：负责操纵表内数据的命令，以及负责管理数据库自身的命令。在网络上有很多介绍全部 SQL 命令以及如何使用这些命令的优秀入门教程。参见 http://www.w3schools.com/sql/default.asp。

准英语语法（English-like Grammar）。当用户创建一条 SQL 语句时，它会向数据库产生一个请求（或“查询”）；该请求是一条语句，其结构类似于一个英语祈使句，譬如“Select your partner”、“Show your stuff”或“Describe that bully”。SQL 语句中的第一个单词是一个表示动作的英语动词，又称作命令，譬如 show、use、select、drop 等。命令后面跟有一组类似于名词的内容，譬如 show database、use database 或 create database。语句中也可含有介词，譬如 in 或 from。例如：

```
show tables in database
select phones from customer_table
```

用户还可以添加条件子句，以便细化查询，譬如：

```
select campanyname from suppliers where supplierid > 20 ;
```

当在一个查询中列出多项内容时，可以像英语一样用逗号隔开它们；例如在下面这个示例中，选中列表中的各个字段都是以逗号隔开的：

```
select companyname, phone, address from suppliers ;
```

如果查询语句很长很棘手的话，读者可能希望把它们键入常用的编辑器中，因为一条查询会在执行完毕后自动消失。如果将查询保存到编辑器的话，读者就能将它剪切并粘贴回 MySQL 浏览器或命令行界面，而不必重新输入它。最重要的是，请读者确保查询的正确性，以免破坏某个重要的数据库。MySQL 提供了一个“test”数据库，专门用于这种验证工作。

以分号终止 SQL 语句。当在 Google 上搜索“SQL 查询”时，位于搜索结果最上面的是一个名叫 thinkgeek.com 的 Web 站点，该站点销售 T 恤、服饰、电子产品、配件以及家庭办公和计算机产品。该网站“SQL 查询”T 恤衫的销售广告写道：

黑色 T 恤衫，上面用白色字体写出下列 SQL 查询语句：“SELECT * FROM user WHERE clue>0”。不幸的是，该语句不能返回任何结果……呃，噢。你看！这条查询末尾少了分号，因为并不是所有人都使用带有相同控制台 /shell 的同一种数据库——毕竟解决问题的方法不止一种嘛。是吗？是的。

```
mysql> show databases
    ->
    -> ;
+-------------+
| Database    |
+-------------+
| mysql       |
| northwind   |
| phpmyadmin  |
| test        |
+-------------+
4 rows in set (0.00 sec)
```

图 B-3　分号缺失和第二提示符

分号是结束每条查询语句的标准方法。某些数据库系统并不要求使用分号，但 MySQL 是要求的（其 USE 和 QUIT 命令除外）。如果用户忘记输入它，就会看到第二个提示符出现，同时会挂起执行，直到用户加上分号为止。正如图 B-3 所示。

命名规范。使用良好的命名规范会让数据库和数据表显得更加易读。

例如，最好能把表的名称设置为复数形式，而把字段 / 列名

设置为单数形式。为什么呢？因为一个名叫“Shippers”的表一般持有多个 Shipper，而字段名则用于描述每一个 Shipper，因此是单数值，譬如“Company_Name”、“Phone”等等。表名或字段名的首字母一般都是大写的。

像“Company_Name”这样的组合式名称往往会通过下划线分隔开来，并且每个部分的首字母都是大写的。

在任何的数据库名中都不允许出现空格和斜杠。

保留字。所有语言都拥有一组保留字，它们对于该语言拥有特殊的含义。本章将用到大多数的保留字。表 B-1 列出了 SQL 的保留字。（有关所有保留字的列表，请参阅 MySQL 文档。）

表 B-1　SQL 保留字

ALTER	JOIN
AND	LEFT JOIN
AS	LIKE
CREATE	LIMIT
CROSS JOIN	ON
DELETE	OR
DROP	ORDER BY
FROM	RIGHT JOIN
FULL JOIN	SELECT
GROUP BY	SET
INSERT	UPDATE
INTO	WHERE

大小写敏感。　如果用户正在使用 UNIX，则数据库名和表名都是大小写敏感的；如果是 Windows 的话则是不敏感的。规范往往都是把数据库及其数据表都以小写方式命名的。

SQL 对大小写则并不敏感。例如，下列 SQL 命令是等效的：

```
show databases;
SHOW DATABASES;
```

尽管 SQL 命令并不区分大小写，但是按照规范的规定，SQL 关键字应当全部大写，而字段名、表名和数据库名只有首字母为大写，以便让命令更为清晰。

```
SELECT * FROM Persons WHERE Firstname='John'
```

如需使用 LIKE 和 NOT LIKE 命令执行模式匹配的话，则在 MySQL 中这些查询的模式将是大小写敏感的。

结果集。结果集（result set）只不过是另一个表，它含有某个 SQL 查询的执行结果。大多数数据库系统甚至允许用户使用函数在结果集上执行操作，譬如 Move-To-First-Record、Get-Record-Content、Move-To-Next-Record 等等。在图 B-4 中显示的结果集是由一条要求 mysql 显示“shippers”表中所有字段的查询语句创建的。

```
mysql> show fields in shippers;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

图 B-4　结果集是由查询产生的一张表

B.1.4　SQL 和数据库

一套数据库服务器可以支持多个数据库。例如，一套 Oracle 或 MySQL 数据库服务器可以提供一个记账数据库、一个人力资源数据库、再加上第三个电子商务应用数据库，等等。为了查看可用的数据库，SQL 提供了 show 命令。

show databases 命令。如需查看数据库服务器上提供了哪些数据库，不妨使用 show databases 命令。尽管不同用户计算机上看到的数据库列表可能不尽相同，但一般都会出现 mysql 和 test 这两

本 PDF 书签由 龙睿·LoRui 制作。
www.LoRui.com

来看看这些东东……

- ✧ Yahoo!疯了！
- ✧ HP笔记本激活Windows 7

域名转让：

- ✓ DaNanPing.com（大南平）
- ✓ 15500.net
- ✓ Gn7c.com
- ✓ DaLongNan.com（大陇南、大龙南）
- ✓ 51perl.com（无忧 Perl）
- ✓ LoRui.com

个数据库，它们是在 mysql 安装时提供的。mysql 数据库是不可或缺的，因为它负责描述用户访问权限；test 数据库则名实相符，负责提供一个练习数据库，以便测试如何使用数据库。

格式

```
SHOW DATABASES;
```

示例 B.1

```
1   mysql> SHOW databases;
    +------------+
    | Database   |
    +------------+
    | mysql      |
    | northwind  |
    | phpmyadmin |
    | test       |
    +------------+
    4 rows in set (0.03 sec)show databases;
```

USE 命令。USE 命令会把指定数据库设置为用户的默认数据库。从这时起，所有的 SQL 命令都将在这个默认数据库上执行。它是少数几个无需以分号结束的命令之一。

格式

```
USE database_name;
```

示例 B.2

```
1   mysql> USE northwind;
    Database changed
```

解释

1. USE 命令把数据库切换到“northwind”。[a] 命令行界面会报告数据库已经切换。

a. 如需下载“northwind”数据库，可访问 http://www.microsoft.com/downloads/ details.aspx?FamilyID=C6661372-8DBE-422B-8676-C632D66C529C&displaylang=EN。

B.1.5 SQL 数据库表

一个数据库里往往含有一张或多张表。每个表都由一个表名来标识，譬如“Customers”或“Orders”。SHOW TABLES IN 命令会显示某个数据库内的所有表，如图 B-5 所示。SELECT * FROM 命令则会列出指定表中的所有字段和行。表中含有行（又称记录）和列（又称字段）。图 B-6 中的表含有三个记录（每个运货者对应一个记录）和三个列（“ShipperID”、“CompanyName”和“Phone”）。

SHOW 和 DESCRIBE 命令。如需查看什么数据类型可以赋予某张表，则可使用 DESCRIBE 命令（仅用于 MySQL），以及 SHOW FIELDS IN 命令，后者是一种 SQL 标准命令。显示的输出内容是每个字段的名称以及对应于各字段值的数据类型，如图 B-7 所示。数据的类型可以是一个可变长字符串（a variable string of characters）、一个日期、一个数字，等等。例如，类型 varchar(40) 表示一个最大宽度为 40 个字符的字段。此外还会显示用于惟一标识各个记录的主键（primary key）。

```
mysql> show tables in northwind;
+-------------------------+
| Tables_in_northwind     |
+-------------------------+
| categories              |
| customercustomerdemo    |
| customerdemographics    |
| customers               |
| employees               |
| employeeterritories     |
| order_details           |
| orders                  |
| products                |
| region                  |
| shippers                |
| suppliers               |
| territories             |
| usstates                |
+-------------------------+
14 rows in set (0.03 sec)
```

图 B-5　显示“northwind”数据库中所有的表

```
mysql> select * from shippers;
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         1 | Speedy Express   | (503) 555-9831 |
|         2 | United Package   | (503) 555-3199 |
|         3 | Federal Shipping | (503) 555-9931 |
+-----------+------------------+----------------+
3 rows in set (0.00 sec)
```

图 B-6　显示指定表中的内容

格式

```
SHOW FIELDS IN table_name;
or
DESCRIBE table-name;
```

图 B-8 显示了更为简短的 DESCRIBE 命令版本。

```
mysql> show fields in customers;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| CustomerID   | varchar(5)  |      | PRI |         |       |
| CompanyName  | varchar(40) |      | MUL |         |       |
| ContactName  | varchar(30) | YES  |     | NULL    |       |
| ContactTitle | varchar(30) | YES  |     | NULL    |       |
| Address      | varchar(60) | YES  |     | NULL    |       |
| City         | varchar(15) | YES  | MUL | NULL    |       |
| Region       | varchar(15) | YES  | MUL | NULL    |       |
| PostalCode   | varchar(10) | YES  | MUL | NULL    |       |
| Country      | varchar(15) | YES  |     | NULL    |       |
| Phone        | varchar(24) | YES  |     | NULL    |       |
| Fax          | varchar(24) | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
11 rows in set (0.05 sec)
```

图 B-7　SQL 的 SHOW FIELD IN 命令

```
mysql> describe shippers;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

图 B-8 MySQL 的 DESCRIBE 命令

B.2 SQL 数据操纵语言（DML）

SQL 是一种非过程型（nonprocedural）语言，它为数据解析操作提供了相关语法，包括刷新记录、插入记录和删除记录的语法。

这些查询和刷新命令共同组成了 SQL 中的数据操纵语言（Data Manipulation Language）部分。本节将介绍下列 SQL 命令：

- SELECT——从一张数据库表中提取数据。
- UPDATE——刷新数据库表中的数据。
- DELETE——从数据库表中删除数据。
- INSERT INTO——向数据库表里插入新数据。

B.2.1 SELECT 命令

最常用的一种 SQL 命令是 SELECT，负责执行一条查询。SELECT 命令用于根据某些标准从一张表中检索数据。它指定了一组以逗号隔开的待检索字段，以及一个指定待访问数据表的 FROM 子句。其执行结果将保存在一个结果表中，又称作结果集。符号 * 用于表示所有的字段。

格式

```
SELECT column_name(s) FROM table_name

Example:
SELECT LastName,FirstName,Address FROM Students;
```

示例 B.3

```
mysql> SELECT CompanyName FROM Shippers;
+------------------+
| CompanyName      |
+------------------+
| Speedy Express   |
| United Package   |
| Federal Shipping |
+------------------+
3 rows in set (0.05 sec)
```

解释

SELECT 命令将从 Shippers 表中检索出字段 CompanyName 中的所有项。在查询之后显示其结果集。

选择指定的列。 如需从 Shippers 表中选择名叫 CompanyName 和 Phone 的列，则可在 SELECT 后面跟上一组以逗号隔开的字段，这些字段含有将要从 Shippers 表中选取的内容。最终得到的表又称作结果集，参见示例 B.4。

示例 B.4

```
mysql> SELECT CompanyName, Phone FROM Shippers;
+------------------+----------------+
| CompanyName      | Phone          |
+------------------+----------------+
| Speedy Express   | (503) 555-9831 |
| United Package   | (503) 555-3199 |
| Federal Shipping | (503) 555-9931 |
+------------------+----------------+
3 rows in set (0.09 sec)
```

选择所有的列。 如需从 Shippers 表中选择所有列，则可使用 * 符号，而不是列名。参见示例 B.5。* 是一个通配符，用于表示所有的字段（列）。

示例 B.5

```
mysql> SELECT * FROM Shippers;
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         1 | Speedy Express   | (503) 555-9831 |
|         2 | United Package   | (503) 555-3199 |
|         3 | Federal Shipping | (503) 555-9931 |
+-----------+------------------+----------------+
3 rows in set (0.06 sec)
```

SELECT DISTINCT 语句。DISTINCT 关键字用于只返回表中不同（distinct）的值。如果每个特定字段上出现了多个重复的值，则 DISTINCT 结果集只会把它们显示为一个值。

下面这个示例中，首先选择 ShipName 列中所有的值，这时会显示超过 800 个记录。不过请注意，如果使用了 DISTINCT 关键字的话，则检索到的记录将锐减至小于 90 个。

格式

```
SELECT DISTINCT column_name(s) FROM table_name
```

示例 B.6

```
SELECT ShipName from Orders
(Partial Output)
| North/South                        |
| Blauer See Delikatessen            |
| Ricardo Adocicados                 |
| Franchi S.p.A.                     |
| Great Lakes Food Market            |
| Reggiani Caseifici                 |
| Hungry Owl All-Night Grocers       |
| Save-a-lot Markets                 |
| LILA-Supermercado                  |
| White Clover Markets               |
| Drachenblut Delikatessen           |
```

```
| Queen Cozinha                        |
| Tortuga Restaurante                  |
| Lehmanns Marktstand                  |
| LILA-Supermercado                    |
| Ernst Handel                         |
| Pericles Comidas clásicas            |
| Simons bistro                        |
| Richter Supermarkt                   |
| Bon app'                             |
| Rattlesnake Canyon Grocery           |
+-------------------------------------+
830 rows in set (0.00 sec)
```

With the DISTINCT keyword, fewer than 90 records are retrieved:

```
SELECT DISTINCT ShipName FROM Orders;
| Océano Atlántico Ltda.               |
| Franchi S.p.A.                       |
| Gourmet Lanchonetes                  |
| Consolidated Holdings                |
| Rancho grande                        |
| Lazy K Kountry Store                 |
| Laughing Bacchus Wine Cellars        |
| Blauer See Delikatessen              |
| North/South                          |
| Cactus Comidas para llevar           |
| Great Lakes Food Market              |
| Maison Dewey                         |
| Trail's Head Gourmet Provisioners    |
| Let's Stop N Shop                    |
```

通过 LIMIT 限制结果集的行数。 如果用户不希望显示某个大型数据库中的所有记录，则可使用 LIMIT 限制打印的行数。例如，数据库 northwind 中的表含有数以千计的记录。在前面的示例中，最好只显示其中少量记录，只需说明查询成功便可。由于用户得到的是部分列表，因此还有可能希望知道表中的记录总数。使用 SQL_CALC_FOUND_ROWS 选项和 SQL 的 FOUND_ROWS() 函数就能做到这一点。SQL 将会算出记录总数，而 FOUND_ROWS() 函数将会允许读者显示此项计算结果。

示例 B.7

```
mysql> select ShipName from Orders LIMIT 10;
+---------------------------+
| ShipName                  |
+---------------------------+
| Vins et alcools Chevalier |
| Toms Spezialitaten        |
| Hanari Carnes             |
| Victuailles en stock      |
| Suprêmes délices          |
| Hanari Carnes             |
| Chop-suey Chinese         |
| Richter Supermarkt        |
| Wellington Importadora    |
| HILARION-Abastos          |
+---------------------------+
```

```
10 rows in set (0.00 sec)
```

解释

通过一个参数（这里是 10），LIMIT 指定了从结果集顶部开始返回的行数。

示例 B.8

```
mysql> SELECT SQL_CALC_FOUND_ROWS ShipName from Orders
    -> LIMIT 5;
+---------------------------+
| ShipName                  |
+---------------------------+
| Vins et alcools Chevalier |
| Toms Spezialitaten        |
| Hanari Carnes             |
| Victuailles en stock      |
| Suprêmes délices          |
+---------------------------+
5 rows in set (0.03 sec)

mysql> SELECT FOUND_ROWS();
+--------------+
| FOUND_ROWS() |
+--------------+
|         830  |
+--------------+
1 row in set (0.03 sec)
```

解释

SQL 会计算记录总数，这里限制为 5；而 FOUND_ROWS() 函数将允许读者显示该项计算结果。

WHERE 子句。 如果需要在某一组特定条件为真时才选择所需字段，那该怎么办呢？例如，需要列出所有来自瑞典并在去年付款超过 50000 美元的顾客。WHERE 子句是可选的，它会根据关键字 WHERE 后面描述的条件来指定应当选择哪些数据值或行。在创建条件（又称选择标准，即 selection criteria）时，SQL 提供了一组运算符，以便进一步指明哪些条件应当出现在 WHERE 子句中。详见表 B-2。

格式

```
SELECT column FROM table WHERE column operator value
Example:
SELECT phone FROM shippers WHERE country like "Sw";
```

表 B-2　SQL 运算符

运算符	描　述	示　例
=	等于	where country = 'Sweden'
<> , !=	不等于[a]	where country <> 'Sweden'
>	大于	where salary > 50000
<	小于	where salary < 50000
>=	大于或等于	where salary >= 50000

（续）

运算符	描　述	示　例
<=	小于或等于	where salary <= 50000
IS [NOT] NULL	是 NULL（非值）或不是 NULL	where birth = NULL
BETWEEN	位于某个范围之间	where last_name BETWEEN 'Dobbins' AND 'Main'
LIKE	搜索类似于模式的值	where last_name LIKE 'D% '
NOT LIKE	搜索不类似于模式值	where country NOT LIKE 'Sw%'
! , NOT	逻辑非，否定	where age ! 10;
|| , OR	逻辑或	where order_number>10 || part_number= 80
&& , AND	逻辑与	where age > 12 && age <21
XOR	异或	where status XOR

a. 在某些版本的 SQL 中，运算符 <> 亦可写成 !=。

使用引号。引号在任何一种编程语言中都是问题的焦点。应当使用单引号还是双引号？应当在什么时候使用引号？

SQL 在文本值上用到了单引号（包括 MySQL 在内的大多数数据库系统也都支持双引号）。而数字值则不应当放在引号中。

对于文本值而言，下面这个示例是正确的：

```
SELECT * FROM Students WHERE FirstName='Macro'
```

而下面这个示例则是错误的：

```
SELECT * FROM Students WHERE FirstName=Macro      Marco 应当改成 "Marco"
```

对于数字值而言，下面这个示例是正确的：

```
SELECT * FROM Students WHERE Year > 2004
```

而下面这个示例则是错误的：

```
SELECT * FROM Students WHERE Year > '2004'              '2004' 应当改为 2004
```

使用 = 和 <> 运算符。在图 B-9 中，如果 WHERE 子句中的条件值为真，就从 Customers 表中检索 CompanyName 和 Phone 字段。也就是说，上述条件是 Country 字段中的字符串值完全等于字符串 Italy（即它们必须含有相同的字符数和字符类型）。运算符 <> 则用于检测“不等于”的情况。

```
mysql> select CompanyName, Phone FROM Customers
    -> WHERE Country='Italy';
+------------------------------+--------------+
| CompanyName                  | Phone        |
+------------------------------+--------------+
| Franchi S.p.A.               | 011-4988260  |
| Magazzini Alimentari Riuniti | 035-640230   |
| Reggiani Caseifici           | 0522-556721  |
+------------------------------+--------------+
3 rows in set (0.00 sec)
```

图 B-9　带有 = 运算符的 WHERE 子句

什么是 NULL？ NULL 意味着字段中没有任何值，或者含有未知的值；但并不是说字段含有零值。如果某个字段值为 NULL，它就是空的；如果不是 NULL，它就含有某个数据。字段的默认值都是 NULL，除非在定义表的时候把它指定为 NOT NULL。详见示例 B.9 和 B.10。

示例 B.9

```
mysql> SELECT region, country FROM suppliers
    -> WHERE region IS NULL;
+--------+-------------+
| region | country     |
+--------+-------------+
| NULL   | UK          |
| NULL   | Japan       |
| NULL   | Japan       |
| NULL   | UK          |
| NULL   | Sweden      |
| NULL   | Brazil      |
| NULL   | Germany     |
| NULL   | Germany     |
| NULL   | Germany     |
| NULL   | Italy       |
| NULL   | Norway      |
| NULL   | Sweden      |
| NULL   | France      |
| NULL   | Singapore   |
| NULL   | Denmark     |
| NULL   | Netherlands |
| NULL   | Finland     |
| NULL   | Italy       |
| NULL   | France      |
| NULL   | France      |
+--------+-------------+
20 rows in set (0.00 sec)
```

解释

从 suppliers 数据库中显示那些 region 值 IS NULL（即不含任何值）的记录的 region 与 country 字段。

示例 B.10

```
mysql> SELECT region, country FROM suppliers
    -> WHERE region NOT NULL;
+----------+-----------+
| region   | country   |
+----------+-----------+
| LA       | USA       |
| MI       | USA       |
| Asturias | Spain     |
| Victoria | Australia |
| OR       | USA       |
| MA       | USA       |
| NSW      | Australia |
| Québec   | Canada    |
| Québec   | Canada    |
+----------+-----------+
9 rows in set (0.00 sec)
```

解释

从 suppliers 数据库中显示那些 region 值 NOT NULL（即含有某个值）的记录的 region 与 country 字段。

> 和 < 运算符。 运算符 > 和 < 用于选取那些字段值大于或小于某个值的行，譬如：

```
SELECT product, price FROM table WHERE price > 50;

SELECT product, price FROM table
WHERE price > 50 && price < 100;
```

读者还可使用 >= 和 <= 运算符来选取那些其字段值大于等于或小于等于某个值的行：

```
SELECT product, price FROM table
WHERE price >=50;
```

示例 B.11

```
mysql> SELECT UnitPrice, Quantity FROM Order_Details
    -> WHERE UnitPrice > 1 && UnitPrice < 3;
+-----------+----------+
| UnitPrice | Quantity |
+-----------+----------+
|    2.0000 |       25 |
|    2.0000 |       60 |
|    2.0000 |       24 |
|    2.0000 |       20 |
|    2.0000 |        8 |
|    2.0000 |       60 |
|    2.0000 |       49 |
|    2.0000 |       50 |
|    2.0000 |       20 |
```

示例 B.12

```
mysql> SELECT CategoryName from categories WHERE CategoryName < 'D';
+--------------+
| CategoryName |
+--------------+
| Beverages    |
| Condiments   |
| Confections  |
+--------------+
3 rows in set (0.00 sec)
```

AND 和 OR 运算符。 运算符 AND 与 OR 用在 WHERE 子句中，负责指明希望从表中选择哪种数据。AND 运算符会检查一个或多个条件，判断是否所有的条件都为真；如果是的话，SELECT 就会显示这些行。只要任意一个列出的条件为真，OR 运算符就会显示该行内容。AND 运算符亦可写作 && 符号，而 OR 运算符也可写成 ||。

示例 B.13

```
mysql> SELECT ContactName FROM Suppliers
        -> WHERE City = 'Montreal' AND Region = 'Quebec';
```

```
+------------------+
| contactname      |
+------------------+
| Jean-Guy Lauzon  |
+------------------+
1 row in set (0.03 sec)
```

解释

当使用 &&（AND）运算符时，位于 WHERE 子句中的所有待测条件都必须为真；也就是说，City 必须是 Montreal，同时 Region 也必须是 Quebec。如果两个条件都为真，则 SELECT 将会从 Suppliers 数据库中打印 ContactName 字段内容。

示例 B.14

```
mysql> SELECT CompanyName, City FROM Suppliers WHERE
    -> City = 'Montreal' OR City = 'Boston';
+------------------------------+----------+
| CompanyName                  | City     |
+------------------------------+----------+
| New England Seafood Cannery  | Boston   |
| Ma Maison                    | Montreal |
+------------------------------+----------+
2 rows in set (0.00 sec)
```

解释

当使用 ||（OR）运算符时，位于 WHERE 子句中的待测条件只要有一个为真即可; 也就是说，当 City 等于 Montreal 或者 City 等于 Boston 时，SELECT 将会从 Suppliers 数据库中打印出 CompanyName 与 City 字段。

LIKE 与 NOT LIKE 条件。 LIKE 模式匹配运算符是一个功能强大的运算符，可以用作 WHERE 子句中的一个条件，允许用户只选择“像是”或者匹配某个模式的行。

百分号符号（%）是一种通配符，能够匹配任何可能出现在指定字符前面和 / 或后面的字符。

_ 可用于匹配单个字符。

LIKE 条件可以用在任何合法的 SQL 语句中，包括 SELECT、INSERT、UPDATE 或 DELETE 语句。

格式

```
SELECT column FROM table WHERE column LIKE pattern
SELECT column FROM table WHERE column NOT LIKE pattern

Example:
SELECT column FROM customer WHERE last_name LIKE 'Mc%';
```

上面这个示例展示了如何在 LIKE 和 NOT LIKE 模式匹配过程中使用通配符 % 和 _。

模式匹配和 % 通配符。通配符 % 用于在执行模式匹配时表示一个或多个任意字符。例如，当搜寻所有区号为 408 的电话号码时，读者可以使用 408%，其中的 % 将会替换为 408 后面的任意字符。

示例 B.15

```
mysql> SELECT CompanyName, Country FROM Customers
```

```
    -> WHERE country like 'Sw%';
+--------------------+-------------+
| CompanyName        | Country     |
+--------------------+-------------+
| Berglunds snabbköp | Sweden      |
| Chop-suey Chinese  | Switzerland |
| Folk och fä HB     | Sweden      |
| Richter Supermarkt | Switzerland |
+--------------------+-------------+
4 rows in set (0.00 sec)
```

解释

SELECT 语句将返回所有来自以 Sw 开头的国家的顾客。

示例 B.16

```
mysql> SELECT City, Country FROM Suppliers WHERE City LIKE '%o';
+-----------+---------+
| City      | Country |
+-----------+---------+
| Tokyo     | Japan   |
| Oviedo    | Spain   |
| Sao Paulo | Brazil  |
| Salerno   | Italy   |
+-----------+---------+
4 rows in set (0.00 sec)
```

解释

SELECT 语句将返回所有满足条件的城市和国家，其中 % 会匹配以字母 o 结尾的任意城市。

示例 B.17

```
mysql> SELECT Companyname FROM customers
       ->     WHERE CompanyName LIKE '%Super%';
+--------------------+
| Companyname        |
+--------------------+
| LILA-Supermercado  |
| Richter Supermarkt |
+--------------------+
2 rows in set (0.00 sec)
```

解释

SELECT 语句将返回所有满足条件的公司名称，其中 % 会匹配任何含有模式 Super 的公司名。

_ 通配符。下面这个示例显示了下划线（_）通配符是如何工作的。请记住，_ 只匹配单个字符。

示例 B.18

```
mysql> SELECT extension, firstname FROM employees
    -> WHERE extension LIKE '4_ _';
+-----------+-----------+
| extension | firstname |
+-----------+-----------+
| 428       | Michael   |
| 465       | Robert    |
| 452       | Anne      |
```

```
+-----------+-----------+
3 rows in set (0.00 sec)
```

解释

这条 SELECT 语句将返回所有满足条件的扩展名和文件名，它们的扩展名含有三个字符并以一个 4 作为首字符。符号 _ 用于匹配单个字符。

BETWEEN 语句。关键字 BETWEEN 允许用户根据一组表示范围的条件来选择所需字段。BETWEEN 子句的语法如下所示：

格式

```
SELECT column  FROM table
WHERE column BETWEEN 'value1' AND 'value2'

Example:
select age from person where age BETWEEN 10 && 20;
```

示例 B.19

```
mysql> SELECT ProductName, ProductId
    -> FROM Products WHERE ProductId BETWEEN 30 AND 33;
+-----------------------+-----------+
| ProductName           | ProductId |
+-----------------------+-----------+
| Nord-Ost Matjeshering |        30 |
| Gorgonzola Telino     |        31 |
| Mascarpone Fabioli    |        32 |
| Geitost               |        33 |
+-----------------------+-----------+
4 rows in set (0.06 sec)
```

解释

SELECT 语句将返回那些 ProductId 值位于 30 和 33 之间的产品名称与产品 ID。

借助 ORDER BY 对结果排序。通过使用 ORDER BY 子句，用户便可以指定的次序显示查询的结果。各行可以按照升序（ASC，默认情况）或降序（DESC）进行排列，用于排序的值既可以是字符串，也可以是数字。

格式

```
SELECT column FROM table
[WHERE condition]
ORDER BY column [ASC, DESC]

Example:
SELECT Company, OrderNumber FROM Orders
ORDER BY Company
```

示例 B.20

```
mysql> SELECT CompanyName, ContactName FROM suppliers
    -> ORDER BY CompanyName LIMIT 10;
```

```
+-------------------------------------+----------------------------+
| CompanyName                         | ContactName                |
+-------------------------------------+----------------------------+
| Aux joyeux ecclésiastiques          | Guylène Nodier             |
| Bigfoot Breweries                   | Cheryl Saylor              |
| Cooperativa de Quesos 'Las Cabras'  | Antonio del Valle Saavedra |
| Escargots Nouveaux                  | Marie Delamare             |
| Exotic Liquids                      | Charlotte Cooper           |
| Forêts d'Trables                    | Chantal Goulet             |
| Formaggi Fortini s.r.l.             | Elio Rossi                 |
| G'day, Mate                         | Wendy Mackenzie            |
| Gai pâturage                        | Eliane Noz                 |
| Grandma Kelly's Homestead           | Regina Murphy              |
+-------------------------------------+----------------------------+
10 rows in set (0.06 sec)
```

解释

以升序方式对 CompanyName 字段进行排序，并将其结果限制为 10 个记录。

示例 B.21

```
mysql> SELECT CompanyName, ContactName FROM suppliers
    -> ORDER BY CompanyName DESC LIMIT 10;
+----------------------------------------+----------------------------+
| CompanyName                            | ContactName                |
+----------------------------------------+----------------------------+
| Zaanse Snoepfabriek                    | Dirk Luchte                |
| Tokyo Traders                          | Yoshi Nagase               |
| Svensk Sjöföda AB                      | Michael Björn              |
| Specialty Biscuits, Ltd.               | Peter Wilson               |
| Refrescos Americanas LTDA              | Carlos Diaz                |
| Plutzer Lebensmittelgro-markte AG      | Martin Bein                |
| PB Knackebröd AB                       | Lars Peterson              |
| Pavlova, Ltd.                          | Ian Devling                |
| Pasta Buttini s.r.l.                   | Giovanni Giudici           |
| Norske Meierier                        | Beate Vileid               |
29 rows in set (0.00 sec)
```

解释

以降序方式对 CompanyName 字段进行排序，并将其结果限制为 10 个记录。

B.2.2 INSERT 命令

INSERT INTO 语句用于把新的行插入表中。在 VALUES 关键字后面，列出了一组以逗号隔开的列名。

格式

```
INSERT INTO table_name VALUES (value1, value2,....)
```

You can also specify the columns for which you want to insert data:

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,....)
```

示例 B.22

```
INSERT INTO Shippers (CompanyName, Phone)
VALUES ('Canada Post', '416-555-1221');
+-----------+------------------+----------------+
| ShipperID | CompanyName      | Phone          |
+-----------+------------------+----------------+
|         1 | Speedy Express   | (503) 555-9831 |
|         2 | United Package   | (503) 555-3199 |
|         3 | Federal Shipping | (503) 555-9931 |
|         4 | Canada Post      | 416-555-1221   |
+-----------+------------------+----------------+
```

解释

INSERT INTO 语句把一个新的行插入到 Shippers 表中。为此，首先列出其字段名，然后在 VALUES 关键字后面列出这些字段的对应值。这里不包含 ShipperID 字段，因为在创建表时就已经把 ShipperID 设置成了 PRIMARY KEY 类型，因此数据库会在每次添加新的运货者记录时自动递增该字段。（让数据库自动递增 PRIMARY KEY 字段能保证它们的值永远是惟一的。）如需查看该表最初是如何创建的，可以查看来自下列 DESCRIBE 命令的输出：

```
mysql> DESCRIBE shippers;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| ShipperID   | int(11)     |      | PRI | NULL    | auto_increment |
| CompanyName | varchar(40) |      |     |         |                |
| Phone       | varchar(24) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
```

B.2.3 UPDATE 命令

UPDATE 语句用于修改表中的数据。用户应在 UPDATE 命令之后列出待修改数据所在的表名，然后是一个负责指明修改哪个字段的 SET 语句，最后则是将要赋予该字段的新值。此外还可使用 WHERE 子句进一步指明修改哪些数据，进而限制更新操作的执行范围。

示例 B.23 的关键之处就是使用 WHERE 语句限制了更新操作的执行范围。

格式

```
UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value

Example:
UPDATE orders SET ShipCountry="Luxembourg" WHERE CustomerId='whitc';
```

示例 B.23

```
1  mysql> select * from shippers;
   +-----------+------------------+-----------------+
   | ShipperID | CompanyName      | Phone           |
   +-----------+------------------+-----------------+
   |         1 | Speedy Express   | (503) 555-9831  |
   |         2 | United Package   | (503) 555-3199  |
   |         3 | Federal Shipping | (503) 555-9931  |
```

```
   +-----------+------------------+----------------+
   3 rows in set (0.00 sec)

2  mysql> UPDATE shippers SET PHONE='(777) 444-1234'
   -> WHERE companyname = 'Federal Shipping';
   Query OK, 1 row affected (0.08 sec)
   Rows matched: 1  Changed: 1  Warnings: 0

3  mysql> select * from shippers;
   +-----------+------------------+----------------+
   | ShipperID | CompanyName      | Phone          |
   +-----------+------------------+----------------+
   |         1 | Speedy Express   | (503) 555-9831 |
   |         2 | United Package   | (503) 555-3199 |
   |         3 | Federal Shipping | (777) 444-1234 |
   +-----------+------------------+----------------+
   3 rows in set (0.00 sec)
```

解释

1. SELECT 命令显示 Shippers 表中所有的字段。
2. UPDATE 命令允许用户更改一个已有的记录。此处更改 Fereral Shipping 的电话号码。
3. 这个 SELECT 命令的输出内容说明前述 UPDATE 命令已经成功地更改了 Fereral Shipping 的电话号码。

B.2.4 DELETE 命令

DELETE 语句用于删除表中的行，并返回删去的行数。DELETE 会使用 FROM 子句来指定含有待删除数据的表名，并通过 WHERE 子句指明筛选条件，以便确定应当删除哪些数据。

请务必小心！如果没有提供 WHERE 子句的话，就会删掉所有的行。[⊖]

如果指定了 ORDER BY 子句，则会按照给定的顺序依次删除各行。LIMIT 子句则会为待删除的行数设定一个限制值。

格式

```
DELETE FROM table_name
WHERE column_name = some_value
```

DELETE 语句非常类似于 UPDATE 语句。如要删除前面所述的记录，读者可以输入如下查询语句：

```
DELETE FROM Shippers WHERE CompanyName='Canada Post';
```

B.3 SQL 数据定义语言

数据定义语言（DDL）是 SQL 的一个组成部分，负责创建或销毁数据库对象。读者还可用它去定义索引（键）、在不同表之间指定链接、并能在不同数据库表之间置入限制。一般而言，诸如创建和销毁数据库之类的操作都是由数据库管理员决定的，此外，他还有权根据获得的访问权限实

⊖ 读者也可设置 MySQL，使得如果在 DELETE 时不带 WHERE 子句，它就不删除任何行。

现表的创建和丢弃（drop）操作。

SQL 中最重要的数据定义语句包括如下几种：

- `CREATE TALBE`——创建新的数据库表。
- `ALTER TABLE`——更改（修改）数据库表。
- `DROP TABLE`——删除数据库表。
- `CREATE INDEX`——创建一个索引（查询键）。
- `DROP INDEX`——删除索引。

B.3.1 创建数据库

创建数据库是一项非常简单的工作。读者所要做的只是运行一条命令，其参数也只有一个数据库名而已。

格式

```
CREATE DATABASE database_name
```

在前面的示例中，我们用到了 northwind 数据库。由于本章将为附录 A 的艺廊实现一个完整的 Web 应用，因此现在需要为该应用创建数据库。

示例 B.24

```
1   mysql> CREATE DATABASE gallerydb;
    Query OK, 1 row affected (0.03 sec)

2   mysql> show databases;
    +-------------+
    | Database    |
    +-------------+
    | gallerydb   |
    | mysql       |
    | northwind   |
    | phpmyadmin  |
    | test        |
    +-------------+
    5 rows in set (0.00 sec)
```

解释

大功告成。现在该数据库已经创建好了。请注意，尽管已经创建了数据库，但目前我们尚不在该数据库中。因此需要使用下面一个示例中的 USE 命令来将这个新建的数据库设置成当前默认数据库。

示例 B.25

```
mysql> USE gallerydb;
Database changed
```

解释

我们现在是在 gallerydb 数据库中，此后所有的 SQL 命令都会在该数据库中执行。

B.3.2 SQL 数据类型

在创建好数据库之后，用户还应添加构成该数据库的各张表。在创建一张表之前，必须首先决

定哪种类型的数据应当保存到表中；例如，一行当中是否应当包括名字、日期、部件号码、社会安全号、价格，等等。数据类型负责指定列持有什么类型的数据。基本的数据类型包括字符串型、数值型、日期与时间型。如需了解完整的类型列表文档，请查阅 http://dev.mysql.com/doc/refman/5.0/en/data-types.html。

表 B-3 含有 SQL 中最为常见的数据类型：

表 B-3 最常见的 SQL 数据类型

数据类型	描 述
数字型	
INTEGER	持有一个长度为 4 个字节的完整数字
INT UNSIGNED	持有一个长度为 4 个字节的非负完整数字
SMALLINT	持有一个长度为 2 个字节的完整数字
TINYINT	持有一个长度为 1 个字节的完整数字
FLOAT(m,d)	一个长度为 4 个字节的小数值。对 999.0009 套用 FLOAT(7,4) 就会得到 999.0001。m 负责指定该数字的最大位数。d 负责指定位于整数部分右侧的小数部分的最大位数
DOUBLE(m,d)	一个长度为 8 个字节的双精度小数值
DECIMAL(m,d)	一个长度为 8 个字节的实数或小数。m 负责指定该数字的最大位数。d 负责指定位于整数部分右侧的最大位数
NUMERIC(m,d)	DECIMAL 和 NUMERIC 数据类型用于以确定的精度保存精确的数值型数据值；例如，资金数据。它持有带小数的数字
字符串型	
CHAR(SIZE)	持有一个长度固定的字符串（可以含有字母、数字和特殊字符），其长度从 0 到 255 个字符。在括号中指定其固定长度
VARCHAR(SIZE)	一个长度可变的字符串（可以含有字母、数字和特殊字符），在 MySQL5.0.3 及其以后版本中，其长度范围是从 0 到 65535 个字符。在括号中指定其最大长度
TINYTEXT	一个最大长度为 255 个字符的字符串
TEXT	一个最大长度为 65535 个字符的长度可变的字符串，用于保存大文本文件、文本区域内容等
BLOB	二进制大对象。它是一种最大长度为 65535 个字符的二进制字符串，用于保存二进制文件、图像、声音等
日期与时间型	
DATE	（yyyy-mm-dd）年、月、日；例如，2006-10-30（注意：MySQL 允许用户存入 0000-00-00，称之为“伪日期”）
.DATETIME	（yyyy-mm-dd hh:mm:ss）日期和时间；例如，2006-10-30 22:59:59
TIMESTAMP	（yyyy-mm-dd hh:mm:ss）日期和时间；例如，1970-01-01（某一行上最后一个事务的日期与时间）
TIME	（hh:mm:ss）时间；例如，10:30:58
YEAR	（yyyy | yy）四位数或两位数形式的年份；例如，1978 或 78

B.3.3 创建表

创建表要比创建数据库稍微复杂一些。CREATE TABLE 语句用于在数据库中创建一张新表。用户首先必须为这个表命名，然后指定表内含有的所有字段、数据类型和其他任何属性。数据的类型

可以是一个整数、一个像 5.4 这样的浮点数（实数）、一个由字符构成的字符串、一个日期、一个时间，等等。并非所有数据库都用相同的方式指定数据类型。如要了解在 MySQL 中可以选用哪些数据类型与属性，可以参见表 B-3 或者 MySQL 文档。

正确设计表结构是一项非常重要的工作。如果读者以前没有接触过数据库的话，就更需要深入地研究这个主题。http://databases.about.com/od/specificproducts/a/normalization.htm 给出了一套数据库设计方面的优秀初学者教程。就本章而言，下面列出了必须在设计数据表时牢记在心的几条规则：

1. 为字段选择正确的数据类型；例如，对主键使用整数型（integer），对较大的数字使用浮点型（float）或双精度浮点型（double），对货币字段使用十进制数（decimal）或数值型（numeric），对时间和日期字段选择正确的日期格式，并且要为字符长度可变的字符串指定充裕的字段宽度，譬如姓名字段和地址字段。如果需要保存二进制数据的话，譬如图像与声音，则应使用支持大批量数据的数据类型，譬如 blob 型和文本（text）型。详见表 B-3。

2. 为各列提供简洁且有意义的列名，并确保这些列名在表中都是惟一的。请不要像下面这样在一个表中出现多个重复的列。这里不应该重复出现三个以 phone 开头的列。

First_Name	Last_Name	Phone1	Phone2	Phone3
Joe	Blow	415-444-3333	333-111-1233	652-345-1123

3. 在每一行的各个列中，只能保存单个值 ；例如，假定有一个 Phone 字段，那就不能把“手机、家庭电话、商务电话”一股脑儿放到该表的单元格中，如下所示：

First_Name	Last_Name	Phone
Joe	Blow	415-444-3333, 333-111-1233, 652-345-1123

4. 为每一组相关的项目分别创建不同的表，并为每一行定义惟一标识它的列（即主键），如下所示：

User Table:

Customer_Id	First_Name	Last_Name
1	Joe	Blow

Phone Table:

Customer_Id	Cell	Business	Home
1	415-444-3333	333-111-1233	652-345-1123

5. 如果还是存在冗余数据的话，最好将它们放入属于自己的表中，并使用外键（foreign key）在表与表中间建立关系（relation）。

格式

```
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
```

```
column_name3 data_type   <-- no comma on the last entry
)
```

示例 B.26

```
1   mysql> CREATE DATABASE pets;
    Query OK, 1 row affected (0.24 sec)
2   mysql> USE pets;
3   mysql> CREATE TABLE dog
    -> ( name varchar(20),
    ->   owner varchar(20),
    ->   breed varchar(20),
    ->   sex char(1),
    ->   birth date,
    ->   death date
    -> );
    Query OK, 0 rows affected (0.16 sec)
4   mysql> describe dog;
    +-------+-------------+------+-----+---------+-------+
    | Field | Type        | Null | Key | Default | Extra |
    +-------+-------------+------+-----+---------+-------+
    | name  | varchar(20) | YES  |     | NULL    |       |
    | owner | varchar(20) | YES  |     | NULL    |       |
    | breed | varchar(20) | YES  |     | NULL    |       |
    | sex   | char(1)     | YES  |     | NULL    |       |
    | birth | date        | YES  |     | NULL    |       |
    | death | date        | YES  |     | NULL    |       |
    +-------+-------------+------+-----+---------+-------+
    6 rows in set (0.00 sec)
```

解释

1. 创建一个名叫 pets 的数据库。
2. 选择并进入 pets 数据库。
3. 创建一个名叫 dogs 的表，并设定其字段和数据类型。其中，name、owner 和 breed 字段将由最大宽度为 20 个字符的字符串组成。sex 字段只有一个字符，其值为 f 或 m，分别表示女性与男性。birth 和 death 列则拥有 date 类型。
4. DESCRIBE 命令类似于 SHOW 命令。它负责显示新表的布局（layout）。

现在，我们便可向新表中插入一些数据了。

示例 B.27

```
mysql> INSERT INTO dog(name,owner,breed, sex, birth, death)
    -> VALUES('Fido','Mr. Jones', 'Mutt', 'M', '2004-11-12',
       '2006-04-02');
Query OK, 1 row affected (0.09 sec)
```

B.3.4 创建键

在现实生活中，人们的身份可以由社会安全号、驾驶执照号和雇员号来标识；图书可以通过 ISBN 号来标识；而 Web 商店订单则可通过一个购买订单号来标识。上述这些标识号都必须是惟一的，因此任意两个人不会拥有相同的社会安全号，任意两本书也不会拥有相同的 ISBN 号，依此类推。

键（key）负责在表中惟一标识一条记录。有两种不同的键：主键（primary key）和外键（foreign key）。

主键（primary key）。 每张表里一般都拥有一个主键。主键用于在数据库中惟一标识一条记录。它们必须是惟一的，从不更改，每张表中仅出现一次，并且一般都是数值型的。

用户可以选择手动为每条记录生成惟一编号，亦可让数据库代劳。如果让数据库负责生成主键的话，它会为每条记录提供惟一的编号，其具体做法是：首先给定一个起始值（譬如 1），然后对每一条新加的记录都将该数字递增 1。即使某条记录后来被删除了，其对应的数字也将永远不再循环使用。数据库会自动递增其内部的计数器，确保每条记录都能获得一个惟一的“键”。

如需将某个字段设定为主键，可使用属性 PRIMARY KEY（field_name），并在字段定义后面使用 AUTO_INCREMENT 属性，以便告诉数据库自动生成惟一的编号，主键不应为空（null）。

下面两个示例显示了一个名叫 categories 的表，其主键名叫 CategoryID。每当向该表中添加新的类别（category）时，都会自动递增该主键。

示例 B.28

```
mysql> USE northwind;
Database changed
mysql> DESCRIBE categories;
+--------------+-------------+------+-----+---------+----------------+
| Field        | Type        | Null | Key | Default | Extra          |
+--------------+-------------+------+-----+---------+----------------+
| CategoryID   | int(11)     |      | PRI | NULL    | auto_increment |
| CategoryName | varchar(15) |      | MUL |         |                |
| Description  | longtext    | YES  |     | NULL    |                |
| Picture      | longblob    | YES  |     | NULL    |                |
+--------------+-------------+------+-----+---------+----------------+
4 rows in set (0.09 sec)
```

解释

CategoryID 是主键，它是一个最长 11 位的整数，其起始值是 NULL（没有值），并会不断递增 1。当第一次把一条记录插入该数据库时，主键值为 1。

示例 B.29

```
mysql> SELECT CategoryID, CategoryName FROM categories;
+------------+----------------+
| CategoryID | CategoryName   |
+------------+----------------+
|          1 | Beverages      |
|          2 | Condiments     |
|          3 | Confections    |
|          4 | Dairy Products |
|          5 | Grains/Cereals |
|          6 | Meat/Poultry   |
|          7 | Produce        |
|          8 | Seafood        |
+------------+----------------+
8 rows in set (0.16 sec)
```

解释

主键是 CategoryID，它用于在这个来自 northwind 数据库的表中惟一标识不同的类别。每当把一个新的类别加入这个表时，CategoryID 都会自动递增 1。

外键（foreign key）。 如果另一张表中引用到了这张表的主键，则称作是外键（foreign key）。外键负责在表和表之间建立相互关系。下面的示例显示了两张表，它们都引用了 CategoryID 键，只不过在一张表中它是主键，而在另一张表中它则是外键。

示例 B.30

```
    mysql> DESCRIBE categories;
    +--------------+-------------+------+-----+---------+----------------+
    | Field        | Type        | Null | Key | Default | Extra          |
    +--------------+-------------+------+-----+---------+----------------+
1   | CategoryID   | int(11)     |      | PRI | NULL    | auto_increment |
    | CategoryName | varchar(15) |      | MUL |         |                |
    | Description  | longtext    | YES  |     | NULL    |                |
    | Picture      | longblob    | YES  |     | NULL    |                |
    +--------------+-------------+------+-----+---------+----------------+
    4 rows in set (0.00 sec)

    mysql> DESCRIBE products;
    +----------------+---------------+-----+-----+--------+---------------+
    | Field          | Type          | Null| Key | Default| Extra         |
    +----------------+---------------+-----+-----+--------+---------------
    | ProductID      | int(11)       |     | PRI | NULL   | auto_increment|
    | ProductName    | varchar(40)   |     | MUL |        |               |
    | SupplierID     | int(11)       | YES | MUL | NULL   |               |
2   | CategoryID     | int(11)       | YES | MUL | NULL   |               |
    | QuantityPerUnit| varchar(20)   | YES |     | NULL   |               |
    | UnitPrice      | decimal(19,4)| YES |     | NULL   |               |
    | UnitsInStock   | smallint(6)   | YES |     | NULL   |               |
    | UnitsOnOrder   | smallint(6)   | YES |     | NULL   |               |
    | ReorderLevel   | smallint(6)   | YES |     | NULL   |               |
    | Discontinued   | tinyint(4)    |     |     | 0      |               |
    +----------------+---------------+-----+-----+--------+---------------+
        10 rows in set (0.00 sec)
```

解释

1. categories 表的主键字段名叫 CategoryID。
2. products 表拥有自己的主键（ProductID），此外还有一个名叫 CategoryID 的外键。如果主键被其他表所引用，则称作是外键。

B.3.5 关系

关系型数据库的主要优势之一就是能在表与表之间建立关系（relation）。简而言之，关系能把一张表中的字段和另一张表中的字段连接起来。这种关系允许用户在数据库中搜索相关的记录。

使用一列或多列值从一张表向另一张表匹配行的操作就是所谓的连接（join）运算。有多种不同类型的连接语句，譬如全连接（full join）、交叉连接（cross join）、左连接（left join），等等。这里我们先介绍一种对两张表的简单连接，称为内连接（inner join）。

表和表之间可以通过键相互关联。正如前面所述，主键是一个列，含有惟一标识各行内容的值。在另一张表中与它匹配的键又称作外键。借助这些键，用户便能在条件满足时把来自不同表的数据绑定起来，而不必重复每张表中的所有数据。

不妨考虑前面的示例 B.30，该示例描述了两张来自 northwind 数据库的表。其中一张表名叫

categories，另一张则名叫 products。CategoryID 是 categories 表中的主键字段，它同时也是 products 表中的一个外键。可以使用 CategoryID 键在这两张表之间建立一个关系。

含有公共键的两张表。如前所述，categories 表和 products 表都含有一个 CategoryID 键，并共享相同的值。这样便可在这两张表之间建立一个关系。

这里不妨创建一个关系，列出所有位于 Seafood 类别中的产品名称。由于 products 表中每一个产品都可归结至 categories 表中八种类别之一，因此这两张表可以通过公有的 CategoryID 字段实现整合。

示例 B.31

```
mysql> SELECT CategoryID, CategoryName FROM categories;
+------------+----------------+
| categoryID | categoryName   |
+------------+----------------+
|          1 | Beverages      |
|          2 | Condiments     |
|          3 | Confections    |
|          4 | Dairy Products |
|          5 | Grains/Cereals |
|          6 | Meat/Poultry   |
|          7 | Produce        |
|          8 | Seafood        |
+------------+----------------+
8 rows in set (0.00 sec)
mysql> SELECT CategoryID, ProductName FROM products;
(Partial Output)
+------------+----------------------------------+
| CategoryID | ProductName                      |
+------------+----------------------------------+
|          1 | Chai                             |
|          1 | Chang                            |
|          2 | Aniseed Syrup                    |
|          2 | Chef Anton's Cajun Seasoning     |
|          2 | Chef Anton's Gumbo Mix           |
|          2 | Grandma's Boysenberry Spread     |
|          7 | Uncle Bob's Organic Dried Pears  |
|          2 | Northwoods Cranberry Sauce       |
|          6 | Mishi Kobe Niku                  |
|          8 | Ikura                            |
|          4 | Queso Cabrales                   |
|          4 | Queso Manchego La Pastora        |
|          8 | Konbu                            |
|          7 | Tofu                             |
|          2 | Genen Shouyu                     |
```

解释

本示例显示了来自 categories 表和 products 表的列。在 categories 表中，CategoryID 是主键，它能惟一标识表内其他的字段。在 products 表中，CategoryID 是外键，并对每一种产品都重复了很多次。

使用完全约束名称和点号来连接数据表。 在查询不止一张表时，可使用点号在列名前面加上其所在的表名，以便避免当两张表都具有相同的表名时发生潜在的冲突。正如示例 B.32 所示。

示例 B.32

```
mysql> SELECT CategoryName, ProductName FROM categories, products
    -> WHERE products.CategoryID = 8 AND categories.CategoryID = 8;
+--------------+---------------------------------+
| CategoryName | ProductName                     |
+--------------+---------------------------------+
| Seafood      | Ikura                           |
| Seafood      | Konbu                           |
| Seafood      | Carnarvon Tigers                |
| Seafood      | Nord-Ost Matjeshering           |
| Seafood      | Inlagd Sill                     |
| Seafood      | Gravad lax                      |
| Seafood      | Boston Crab Meat                |
| Seafood      | Jack's New England Clam Chowder |
| Seafood      | Rogede sild                     |
| Seafood      | Spegesild                       |
| Seafood      | Escargots de Bourgogne          |
| Seafood      | Röd Kaviar                      |
+--------------+---------------------------------+
12 rows in set (0.00 sec)
```

解释

在 SELECT 语句中，借助 CategoryID 字段连接两张表（以逗号隔开）。由于该字段名在两张表中一模一样，因此要在字段名前面通过点号加上表名，形如 products.CategoryID 和 categories.CategoryID。在 WHERE 子句中，如果两张表都含有一个值等于 8 的 CategoryID 字段，则连接这两张表。

别名。为了略微简化一下复杂查询语句的输入工作，SQL 提供了一套别名机制，允许用户为列和表指定符号化的名称。别名是由 AS 关键字定义的，它可以由单个字符或缩略的字符串构成。当在 WHERE 子句中通过别名来表示表名时，必须在前面加上点号，以及从该表中选取的字段名称。

格式

```
(Column Alias)
SELECT column_name AS column_alias_name
FROM table_name

(Table Alias)

SELECT column_name
FROM table_name AS table_alias_name
```

示例 B.33

```
mysql> SELECT CategoryName as Foods FROM categories;
+----------------+
| Foods          |
+----------------+
| Beverages      |
| Condiments     |
| Confections    |
| Dairy Products |
| Grains/Cereals |
| Meat/Poultry   |
| Produce        |
```

```
| Seafood         |
+-----------------+
8 rows in set (0.00 sec)
```

解释

来自 categories 表的列名是 CategoryName。在 CategoryName 之后，使用 AS 关键字创建一个名叫 Foods 的别名。此时，当 SELECT 语句返回一个结果集时，其输出会把 Foods 当作这个列的列名。

示例 B.34

```
mysql> SELECT ProductName FROM products AS p, categories AS c WHERE
    -> p.CategoryID = c.CategoryID AND c.CategoryName="SeaFood";
+----------------------------------+
| ProductName                      |
+----------------------------------+
| Ikura                            |
| Konbu                            |
| Carnarvon Tigers                 |
| Nord-Ost Matjeshering            |
| Inlagd Sill                      |
| Gravad lax                       |
| Boston Crab Meat                 |
| Jack's New England Clam Chowder  |
| Rogede sild                      |
| Spegesild                        |
| Escargots de Bourgogne           |
| Röd Kaviar                       |
+----------------------------------+
12 rows in set (0.00 sec)
```

解释

本示例乍一看有点奇怪。为 products 表赋予一个别名 p，同时为 categories 表赋予别名 c。这两个别名都很简短，当需要在查询中涉及不止一张表的时候，它们会让键入过程变得更为方便；例如，无需再键入 products.CategoryID，而是可以键入 p.CategoryID；此外，categories.CategoryID 亦可引用为 c.CategoryID。

B.3.6 更改表

当用户更改（alter）一张表时，会通过添加或丢弃新的列、键、索引和数据表来重新定义其结构。用户还可使用 ALTER 命令去更改列名、类型乃至表名。

格式

```
ALTER TABLE tablename
ADD column datatype

Example:
alter table artist add column ArtDate date;
alter table artist drop column "Address";
```

示例 B.35

```
    use pets;
```

```
1   mysql> ALTER TABLE dog ADD pet_id int(11);
    Query OK, 0 rows affected (0.13 sec)
    Records: 0  Duplicates: 0  Warnings: 0
2   mysql> ALTER TABLE dog MODIFY column pet_id int(11)
    -->  auto_increment primary key;
    Query OK, 1 row affected (0.11 sec)
    Records: 1  Duplicates: 0  Warnings: 0
3   mysql> describe dog;
    +--------+-------------+------+-----+---------+----------------+
    | Field  | Type        | Null | Key | Default | Extra          |
    +--------+-------------+------+-----+---------+----------------+
    | name   | varchar(20) | YES  |     | NULL    |                |
    | owner  | varchar(20) | YES  |     | NULL    |                |
    | breed  | varchar(20) | YES  |     | NULL    |                |
    | sex    | char(1)     | YES  |     | NULL    |                |
    | birth  | date        | YES  |     | NULL    |                |
    | death  | date        | YES  |     | NULL    |                |
    | pet_id | int(11)     |      | PRI | NULL    | auto_increment |
    +--------+-------------+------+-----+---------+----------------+
    7 rows in set (0.00 sec)
    mysql> select * from dog;
+-------+-------------+-------+-----+-----------+-----------+--------+
| name  | owner       | breed | sex | birth     | death     | pet_id |
+-------+-------------+-------+-----+-----------+-----------+--------
| Fido  | Mr. Jones   | Mutt  | M   | 2004-11-12| 2006-04-02|      1 |
| Lassie| Tommy Rettig| Collie| F   | 2006-01-10| NULL      |      2 |
+-------+-------------+-------+-----+-----------+-----------+--------+
2 rows in set (0.00 sec)
```

解释

1. ALTER 命令将通过添加一个名叫 pet_id 的新字段来更改数据表，该字段是一个 11 位整数。
2. 在创建好 pet_id 字段之后，可以再次使用 ALTER 命令来将它设置为主键。每当添加一条新记录时，它都会自动进行递增。
3. 在更改之后，用 DESCRIBE 命令显示表的结构。可见已经添加了一个主键。

B.3.7 删除表

表的删除（drop）操作相对较为简单。只需使用 drop 命令和表名即可：

```
mysql> drop table dog;
Query OK, 20 rows affected (0.11 sec)
```

B.3.8 删除数据库

如需删除某个数据库，可使用 drop database 命令：

```
mysql> drop database pets;
Query OK, 1 row affected (0.45 sec)
```

B.4 SQL 函数

下面这些函数用于对 SQL 查询的输出进行更改或者格式化。这些函数涉及字符串、数字、日期、

服务器与信息等诸多方面。它们会返回一个结果集。函数是与开发商相关的，也就是说 MySQL 支持的函数可能无法得到 Microsoft SQL Server 的支持。如需了解 MySQL 支持的所有函数列表，请阅读 MySQL 文档。

当在 SELECT 语句中使用函数时，该函数将会在调用时显示结果集中相应的列名，正如示例 B.36 所示。

示例 B.36

```
1   mysql> SELECT avg(UnitPrice)
    FROM order_details;
    +-----------------+
    | avg(UnitPrice) |
    +-----------------+
    |    26.21851972 |
    +-----------------+
    1 row in set (0.01 sec)

2   mysql> SELECT avg(UnitPrice) as 'Average Price'
    FROM order_details;
    +----------------+
    | Average Price |
    +----------------+
    |   26.21851972 |
    +----------------+
    1 row in set (0.00 sec)
```

解释

1. 将函数显示成列名。
2. 用户可使用 AS 关键字为结果集中显示函数的那一列创建别名或其他名称。

B.4.1 数值函数

假设用户需要获得所有订单的总额、或者一组项目的平均花销、又或者需要基于某个特定条件获取表内的所有行数。聚集（aggregate）函数能够基于一组其他值返回单个值。如果需要把该函数用在 SELECT 语句项目列表内的其他表达式中，则该 SELECT 语句必须含有一个 GROUP BY 从句。如果聚集函数是 SELECT 语句检索的惟一值，则无需 GROUP BY 子句。表 B-4 列出了这些函数及其语法。

表 B-4 聚集函数

函　数	用　途
`AVG()`	计算并返回某一列的平均值
`COUNT(`*`expression`*`)`	计算表达式定义的行数
`COUNT()`	计算表中所有行的行数
`MIN()`	返回某一列中的最小值
`MAX()`	返回某一列中的最大值
`SUM()`	返回某一列中所有值的和

示例 B.37

```
1   mysql> select count(*) from products;
    +----------+
    | count(*) |
    +----------+
    |       81 |
    +----------+
    1 row in set (0.00 sec)

    mysql> SELECT count(*) as 'Number of Rows' FROM products;
    +----------------+
    | Number of Rows |
    +----------------+
    |             81 |
    +----------------+
    1 row in set (0.00 sec)
```

解释

1. COUNT() 函数负责计算表中所有行的行数。

示例 B.38

```
1   mysql> SELECT avg(UnitPrice)
    FROM order_details;
    +----------------+
    | avg(UnitPrice) |
    +----------------+
    |    26.21851972 |
    +----------------+
    1 row in set (0.01 sec)

2   mysql> SELECT FORMAT(avg(UnitPrice),2) as 'Average Price'
    FROM  order_details;
    +---------------+
    | Average Price |
    +---------------+
    | 26.22         |
    +---------------+
    1 row in set (0.00 sec)
```

解释

1. AVG() 函数能计算并返回某一列的平均值，即 UnitPrice。
2. FORMAT 函数以两位小数的精度返回 AVG() 函数的计算结果。

使用 GROUP BY。GROUP BY 子句用在 SELECT 语句中，负责跨记录收集所有的数据，并根据一列或多列内容组织结果。在使用聚集函数（如 SUM、COUNT、MIN 或 MAX）时，该子句就显得非常管用。请看下面两个示例。

示例 B.39

```
mysql> select CategoryID, SUM(UnitsInStock) as 'Total Units in Stock'
    -> FROM products
    -> GROUP BY CategoryID;
```

```
+------------+---------------------+
| CategoryID | Total Units in Stock |
+------------+---------------------+
|       NULL |                   0 |
|          1 |                 559 |
|          2 |                 507 |
|          3 |                 386 |
|          4 |                 393 |
|          5 |                 308 |
|          6 |                 165 |
|          7 |                 100 |
|          8 |                 701 |
+------------+---------------------+
9 rows in set (0.00 sec)
```

示例 B.40

```
mysql> select C.CategoryName,
    -> SUM(P.unitsInsStock) AS Units
    -> FROM products as P
    -> join categories AS C ON C.CategoryID=
    -> P.CategoryID Group By C.CategoryName;
+----------------+-------+
| CategoryName   | Units |
+----------------+-------+
| Beverages      |   559 |
| Condiments     |   507 |
| Confections    |   386 |
| Dairy Products |   393 |
| Grains/Cereals |   308 |
| Meat/Poultry   |   165 |
| Produce        |   100 |
| Seafood        |   701 |
+----------------+-------+
8 rows in set (0.00 sec)
```

B.4.2 字符串函数

SQL 提供了一系列基本的字符串函数，列于表 B-5 中。

表 B-5 MySQL 字符串函数

函 数	用 途
CONCAT(string1, string2, ...)[a]	将字段值或变量组合到一个字符串中
LOWER(string)	将字符串转换为全小写形式
SUBSTRING(string, position)	抽取字符串中的一部分内容（详见示例 B.41）
TRANSLATE	将字符串从某种字符集切换到另一种字符集
TRIM(' string ');	从字符串中移除首字符、尾字符，或者二者都移除
UPPER(string)	将字符串转换为全大写形式（详见示例 B.41）

a. SQL99 定义了字符串组合运算符（||），可以和 CONCATENATE() 函数配合使用。MySQL 则使用表 B-5 中列出的 concat() 函数。

示例 B.41

```
mysql> select upper(CompanyName) as 'Company' from shippers;
+------------------+
| Company          |
+------------------+
| SPEEDY EXPRESS   |
| UNITED PACKAGE   |
| FEDERAL SHIPPING |
+------------------+
3 rows in set (0.00 sec)

mysql> select lower(CompanyName) as 'Company' FROM shippers;
+------------------+
| Company          |
+------------------+
| speedy express   |
| united package   |
| federal shipping |
+------------------+
3 rows in set (0.00 sec)
```

B.4.3 日期与时间函数

为获取日期与时间值，MySQL 提供了表 B-6 列出的一系列函数。

表 B-6 MySQL 日期与时间函数

函　数	示　例
NOW()	select NOW() ; --> 2006-03-23 20:52:58 (详见示例 B.42)
CURDATE()	select CURDATE () ; --> '2006-12-15' (详见示例 B.42)
CURTIME()	select CURTIME () ; --> '23:50:26' (详见示例 B.42)
DAYOFYEAR(date)	select DAYOFYEAR('2006-12-15'); --> 349
DAYOFMONTH(date)	select DAYOFMONTH ('2006-12-15'); --> 15
DAYOFWEEK(date)	select DAYOFWEEK ('2006-12-15'); --> 6
WEEKDAY(date)	select WEEKDAY ('2006-12-15'); --> 4
MONTHNAME(date)	select MONTHNAME ('2006-12-15'); --> December
DAYNAME(date)	select DAYNAME ('2006-12-15'); --> Friday
YEAR(date)	select YEAR ('2006-12-15'); --> 2006
QUARTER(date)	select QUARTER ('2006-12-15'); --> 4

示例 B.42

```
mysql> select NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2006-03-21 00:32:37 |
+---------------------+
1 row in set (0.00 sec)

mysql> select CURDATE();
+----------------+
|   CURDATE()    |
+----------------+
| 2006-03-21     |
+----------------+
1 row in set (0.03 sec)

mysql> select CURTIME();
+----------------+
|   CURTIME()    |
+----------------+
| 00:12:46       |
+----------------+
1 row in set (0.01 sec)
```

格式化日期与时间。当从表中检索日期与时间时，往往需要对结果进行格式化。例如，当从 northwind 数据库内的 orders 表中选取订单日期时，其结果对用户而言是不大友好的。SQL 中的日期值通常保存为 MM/DD/YY(YY) 的格式。因此这里提供了 DATE_FORMAT() 和 TIME_FORMAT() 函数（详见示例 B.43），它带有一系列参数（详见表 B-7），能够指定应当如何显示输出内容。

示例 B.43

```
mysql> select DATE_FORMAT('2006-03-23', '%W %M %d, %Y') as Today;
+-------------------------+
| Today                   |
+-------------------------+
| Thursday March 23, 2006 |
+-------------------------+
1 row in set (0.00 sec)
mysql> select DATE_FORMAT(OrderDate,'%M %e, %Y - %l:%i %p')
             FROM orders LIMIT 5;

+---------------------------------------------+
| DATE_FORMAT(OrderDate,'%M %e, %Y - %l:%i %p') |
+---------------------------------------------+
| July 4, 1996 - 12:00 AM                      |
| July 5, 1996 - 12:00 AM                      |
| July 8, 1996 - 12:00 AM                      |
| July 8, 1996 - 12:00 AM                      |
| July 9, 1996 - 12:00 AM                      |
+---------------------------------------------+
5 rows in set (0.00 sec)
```

表 B-7　DATE_FORMAT() 和 TIME_FORMAT()

参数	含　义
%a	对星期几的缩写（Sun、Mon、Tues 等）
%b	对月份名的缩写（Jan、Feb、Mar 等）
%c	月份（1-12）
%d	两位数形式的月份内的日数，（01-31）
%D	带有后缀的日数（30th、31st）
%e	月份内的日数（1-31）
%f	微秒（000000..999999）
%H	小时（00..23）
%h	小时（01..12）
%i	分钟，数值型（00..59）
%I	小时（01-12）
%j	一年内的日数（001-366）
%k	小时（0..23）
%l	小时（1-12）
%m	带有前导 0 的月份数（01、06 等）
%M	月份名称（March、April、May 等）
%p	AM/PM
%r	时间，12 小时制（hh:mm:ss，后面是 AM 或 PM）
%S	秒（00..59）
%s	秒（00..59）
%T	时间，24 小时制（hh:mm:ss）
%U	以 Sunday 开头的星期数（00..53）
%u	以 Monday 开头的星期数（00..53）
%v	以 Monday 开头的星期数（01..53）
%V	以 Sunday 开头的星期数（01 .53）
%W	星期几（Sunday、Monday 等）
%w	一周内的天数（0=Sunday..6=Saturday）
%Y	年份（1999、2007）
%y	两位数形式的年份（99、07）
%%	% 文法字符

MySQL EXTRACT 命令。EXTRACT 命令是 MySQL 扩展的示例之一，在 SQL 标准中并没有

定义。它允许用户抽取日期或时间中的不同组成部分，如表 B-8 所示。

表 B-8 日期与时间部分

类 型	格 式
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
MONTH	MONTHS
YEAR	YEARS（详见示例 B.44）
MINUTE_SECOND	“MINUTES:SECONDS”
HOUR_MINUTE	“HOURS:MINUTES”
DAY_HOUR	“DAYS HOURS”
YEAR_MONTH	“YEARS-MONTHS”
HOUR_SECOND	“HOURS:MINUTES:SECONDS”
DAY_MINUTE	“DAYS HOURS:MINUTES”
DAY_SECOND	“DAYS HOURS:MINUTES:SECONDS”

示例 B.44

```
mysql> select EXTRACT(YEAR FROM NOW());
+--------------------------+
| EXTRACT(YEAR FROM NOW()) |
+--------------------------+
|                     2006 |
+--------------------------+
1 row in set (0.03 sec)
```

B.5 附录小结

在本附录中，读者学习了如何使用 SQL 语言创建数据库样式（schema），以及如何对数据库中的记录进行插入、更新（update）、检索（retrieve）、更改（alter）和删除操作。

B.6 读者应当学到的知识

在学习完本附录之后，读者应当能够回答下面这几个问题：

1. 如何检索数据库表中所有的记录？
2. 如何基于指定条件从表中检索一组选定的记录，或检索单条记录？
3. 如何在数据库中选择记录并对它们进行排序？

4. 如何从数据库中选取给定范围内的行？
5. 如何创建一个数据库？
6. 如何创建数据库表？
7. 如何把一个主键赋予某个字段？
8. 记录是怎样插入到数据库表中的？
9. 在表中是如何更新记录的？
10. 如何删除一条记录？

练习 B

1. 进入 MySQL 控制台，使用 show 命令列出全部的数据库。然后选用数据库 mysql，并显示其中所有的表。
2. 创建一个名叫 school 的新数据库。在创建完毕后，读者还可这样使用它：

```
use school;
```

3. 创建一个名叫 student 的数据表，该表由如下字段组成：

```
FirstName
LastName
Email
CellPhone
Major
GPA
StartDate
StudentId
```
(the primary key)

下面列出的数据类型将用于数据表的定义。读者不妨在 Web 上找一个与它相似的数据表作为参考。

```
integer(size)
int(size)
smallint(size)
tinyint(size)
```
Holds integers only

其中各个数字的最大宽度是由位于小括号内的数值来指定的。

```
decimal(size, d)
numeric(size, d)
```
Holds numbers with fractions.

其中各个数字的最大宽度由 size 指定，而小数点后的最大宽度则由 d 确定。

`char(size)`	Holds a fixed-length string (can contain letters, numbers, and special characters). The fixed size is specified by size in parentheses.
`varchar(size)`	Holds a variable-length string (can contain letters, numbers, and special characters). The maximum size is specified by size in parentheses.
`date(yyyymmdd)`	Holds a date.

4. 请使用 SQL 的 describe 语句显示用于创建 school 数据库的所有信息。
5. 请将下列三行插入数据表中：

Row 1: FirstName: John
LastName: Doe
Email: johndoe@smileyface.edu

CellPhone: 408-333-3456
Major: CIS
GPA: 2.8
StartDate: 09/22/2004 (use the correct date format!)
StudentId: 1

Row 2: FirstName: Mary
LastName: Chin
Email: mchin@qmail.com
CellPhone: 408-204-1234
Major: Biology
GPA: 3.3
StartDate: 06/22/2003
StudentId: 2

Row 3: FirstName: Sadish
LastName: Pamel
Email: sadi@univ_ab.edu
CellPhone: 415-204-1234
Major: CIS
GPA: 3.9
StartDate: 06/22/2003
StudentId: 2

6. 请使用 show 命令显示所有字段名。
7. 请使用 select 语句显示下列内容（请将查询语句写在空白行中）:
   a. 所有列中的数据。
   b. 各个学生的名和姓氏。
   c. 某个学生的名、姓氏和专业。
   d. 某个学生的手机号和电子邮件地址。
   e. 所有不同的专业。
   f. 只显示两行内容。
8. a. 选取 GPA 高于 3.0 的所有学生。
   b. 选取 GPA 位于 3.0 与 4.0 之间的所有学生。
   c. 选取手机号码区号是 408 的所有学生。
   d. 显示所有 2003 级学生所在的行。
   e. 选取所有专业是 GIS 且 GPA 高于 3.5 的学生的名字和姓氏。
   f. 选取电子邮件地址以 .com 结尾的电子邮件地址和学生名字。
9. a. 向数据表中插入新的一项。
   b. 根据姓氏对学生数据表进行排序。
   c. 按照 GPA 降序对学生数据表进行排序。
10. 把 Mary 的电话号码更改为 650-123-4563。

下面这三个问题是有关 SQL 函数的：

11. 对所有学生求平均 GPA。
12. 获得数据表中的总行数。
13. 借助 SQL 函数打印当天日期。

附录 C　Perl 和生物学

C.1　什么是生物信息学

“基因数据每天都在雪崩式地增加。这也带来了新的挑战，即如何利用这些海量数据探究 DNA 和蛋白质的相互作用机制，并创建复杂、动态的生命系统。大规模的系统研究（即功能基因学）将在本世纪乃至以后的时间里成为生物学探险的焦点所在……” http://www.ornl.gov/sci/techresources/Human_Genome/project/。生物信息学涉及在计算机上处理上述大规模的生物数据。

在过去 20 年间，特别是在 2003 年之后，计算机已经在生物研究工作中变得越来越重要了。在这一年，人类基因组项目完成了目标，确认了人类 DNA 中的所有基因。（有关该项目的目标，请参阅 http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml。）现在，程序可以快速地分析大量数据，而 Perl 业已成为生物计算领域最流行的语言之一（如果不是最流行的话），负责完成诸如访问本地或远程数据库中的序列数据、传输序列文件、处理序列数据、取得统计信息、对生物系统建立模型之类的任务。

许多生物学家都喜欢编写自己的函数，以便处理海量的数据，并会发现下载和使用 BioPerl 模块将能加速其程序开发过程。并非所有的 Perl 程序员都非常了解生物学知识，正如不是所有的生物学家都会 Perl 一样。因此，下面的概述部分将简单介绍一些生物学知识，并介绍使用 Perl 来处理表示 DNA 序列和蛋白质的符号的几种途径，包括依赖和不依赖于 BioPerl 模块的途径。

C.2　DNA 背景知识

DNA 是产生任何生物的素材。它含有基因指令，能够创造像你我这样的芸芸大众。它还负责产生蛋白质。蛋白质是一种备受关注的物质，因为它能控制细胞的行为，并控制细胞如何工作；譬如，如何消化食物、消除感染、把氧气输送到肌肉与器官，等等。

在使用 Perl 处理 DNA 时，读者将碰到下列词汇：

1. 核苷酸（Nucleotide），或碱基（base）
2. 碱基对（basic pair）
3. 反向互补（reverse complement）
4. 序列（sequencing）

5. 蛋白质与氨基酸
6. RNA 分子和转录
7. BLAST 和 FASTA

DNA 出现在细胞核里面，其结构称作双螺旋结构，因为它的形状像是一条螺旋的梯子，由上千个基因组成。其中每一个梯级都由四种化学物质组成，称作核苷酸（nucleotide），又称碱基（base），其中带有建造身体并维持运行所用到的信息。这四种碱基都分别以单个字母来命名：G（鸟嘌呤）、A（腺嘌呤）、T（胸腺嘧啶）、C（胞嘧啶）。细胞中的这些字母构成了人类基因，它们一个连着一个，组成了一套完备的指令集合，编码出地球上的所有生物。

在一个完整的 DNA 螺旋中，通常由一个氢键来连接 A 和 T，以及 G 和 C。在建造新的细胞时，DNA 字母表中的不同字母将会组合到一起；不过只需这四个字母，DNA 字母表就能拼写出建造新细胞所需要的一切信息。（如果只有一条 DNA 链，则相应的另一条 DNA 链将是该链的反向互补序列。）DNA 碱基的排列顺序又称作序列（sequence）。DNA 中四种碱基的序列足以拼写出所有的指令，这些指令能构建出人体上所有的细胞，并决定每个人的遗传特征。“如果把一个细胞中的所有碱基都写下来的话，这些 A、T、G、C 将足以填满 1000 本电话簿。如果科学家们需要从整个基因中定位出某一小块 DNA 的话，就不得不在数十亿碱基中苦苦搜寻所需的部分！有时这项工作会耗费多年时光。”（摘自 http://www.thetech.org/exhibits/online/genome。）Genbank，即基因序列数据银行（http://www.ncbi.nlm.nih.gov/Genbank），保存有绝大多数已知的序列数据。



图 C-1　双螺旋结构与核苷酸，或碱基

什么是 RNA ？ RNA 类似于 DNA，但它只有一条序列，而不是两条；它也并不一定位于细胞核中。RNA 的配对模式与 DNA 基本相同，除了将 T（胸腺嘧啶）替换成了 U（尿嘧啶）。RNA 分为好几种；其中一种起信使作用，负责把 DNA 编码信息从细胞核外运输到消息中心（message center），后者是细胞中负责根据这些信息生成蛋白质的区域。蛋白质是由形状各异的名叫氨基酸的分子所组成的长链。当食用食物时，身体会消化这些食物，并把它们分解为氨基酸，然后由细胞重新使用这些氨基酸。转运 RNA（transfer RNA）会帮助把编码指令转译为氨基酸序列。“转录（transcription）”这个词专门用于描述 DNA 生成 RNA（核糖核酸）的过程，而“翻译”一词指的是 RNA 将编码翻译成蛋白质的过程。

BLAST（Basic Local Alignment Search Tool）是当前最流行的生物搜索工具之一。它通过发出一个查询请求并在已知序列库中检索的方式来检测目标序列。它会提供一个含有分值的文件，显示该序列在统计学意义上的匹配情况。FASTA 文件是一种简单的文本文件，其第一行以字符“>”开头，其后是所生成 DNA 或基因的名字，再下面各行内容则是核苷酸或氨基酸序列数据。

在查看本附录中或 Web 上的示例时，了解上述这些术语将是非常有益的。

C.3　一些 Perl 示例

下面的示例非常基础。它们演示了如何使用简单的 Perl 程序进行处理，包括使用 tr 函数求一个

序列的反向互补序列，如何使用正则表达式从文件中搜索某种模式或样式，以及如何在循环中使用正则表达式来从一个序列筛选出其碱基。

示例 C.1

```
# Sequence--Reverse Complement
$dna_strand='GGGGaaaaaaaattAtAtat';
$dna_strand=reverse($dna_strand);
print "The original DNA strand is $dna_strand\n";
$dna_strand =~ tr/ACGTacgt/TGCAtgca/;

print "The reversed complement is ",$dna_strand,"\n";

(Output)
The original DNA strand is tatAtAttaaaaaaaaGGGG
The reversed complement is ataTaTaattttttttCCCC
```

示例 C.2

```
File name: fasta
>testsrings
MTKKIGLFYGTQTGKTESVAEIIRDEFGNDVVTLHDVSQAEVTDLNDYQYLIIGCPTWNI
GELQSDWEGLYSELDDVDFNGKLVAYFGTGDQIGYADNFQDAIGILEEKISQRGGKTVGY
WSTDGYDFNDSKALRNGKFVGLALDEDNQSDLTDDRIKSWVAQLKSEFG

(Perl script)
 # Finding a pattern/motif in a protein file
open(PFH, "fasta") or die "Can't open file: $!\n";
$pattern="QTGK";
while($string=<PFH>){
   print "$pattern found on line $..\n"
            if $string =~/$pattern/i;
}
close(PFH);

(Output)
QTGK found on line 1.
```

示例 C.3

```
# Filter sequence from any unwanted characters
# Prints just nucleotides
open(JF, "junkseq") or die;

@line=<JF>;
@chars = split(//,$line[0]);
print "Original sequence with junk\n";
print @chars,"\n\n";
print "Cleaned up sequence\n";
 for($i=0; $i <= $#chars; $i++){
        if($chars[$i] =~ /[Aa]|[Tt]|[Gg]|[Cc]/){
               print uc "$chars[$i]";
        }
}
print "\n";
```

```
(Output)
Original sequence with junk
tg^c*ttcgh#ittgcatgggttc'tt:igg!tt~8$gttcggsstt$$@^ucgte+2%%&tagc

Cleaned up sequence


TGCTTCGTTGCATGGGTTCTTGGTTGTTCGGTTCGTTAGC
```

C.4 什么是 BioPerl

BioPerl 是一种开放源码的（当然也是免费的）工具包，它含有超过 500 种 Perl 模块，能让脚本为基于 Web 的系统分析海量的数据。“BioPerl 提供了可重用的 Perl 模块，方便了 Perl 脚本的编写，它能实现序列的处理、以多种数据格式访问数据库、执行并解析多种分子生物学程序的结果，包括 Blast、clustalw、TCoffee、genscan、ESTscan 以及 HMMER。”BioPerl 还支持对远程数据库的访问，并能为访问本地数据库创建索引。BioPerl 允许用户构建出能够分析海量序列数据的程序。它需要开发者熟悉面向对象的 Perl，并不提供完全可用的程序；它提供的是一组工具，从而简化公用的生物信息编码工作。BioPerl 通常都是由大量程序员共同开发和维护的。

如需了解完整的安装指南，请读者查看 BioPerl 主页 www.bioperl.org。最简便的 BioPerl 安装方式是使用 ActiveState 为各种主流操作系统提供的 PPM 管理器，或者解压 tar 文件并键入如下命令：

```
   >gunzip bioperl-1.2.tar.gz
>tar xvf bioperl-1.2.tar
>cd bioperl-1.2
```

然后调用 make 命令：

```
>perl Makefile.PL
>make
>make test
```

如需下载该模块或获取有关它的信息，请访问如图 C-2 所示 BioPerl Web 站点。

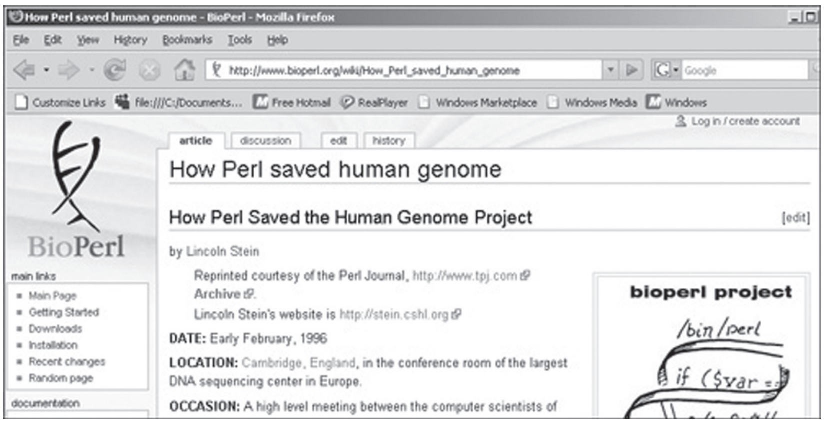


图 C-2　借助 PPM（ActiveState）安装 bioperl

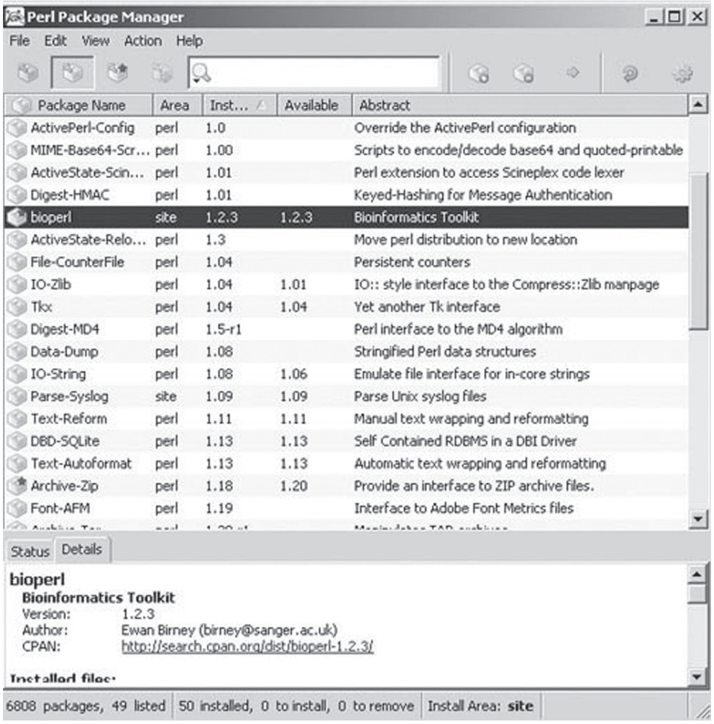


图 C-3 bioperl 是一组模块，这里显示了它在 Windows32 系统上的内容

下面列出了目录 C:\ActivePerl\site\lib\Bio 的内容：

```
07/03/2007  07:14 PM    <DIR>          .
07/03/2007  07:14 PM    <DIR>          ..
07/03/2007  07:14 PM    <DIR>          Align
07/03/2007  07:14 PM    <DIR>          AlignIO
07/03/2007  07:14 PM            14,301 AlignIO.pm
07/03/2007  07:14 PM            23,350 AnalysisI.pm
07/03/2007  07:14 PM             6,042 AnalysisParserI.pm
07/03/2007  07:14 PM             6,422 AnalysisResultI.pm
07/03/2007  07:14 PM             2,798 AnnotatableI.pm
07/03/2007  07:14 PM    <DIR>          Annotation
07/03/2007  07:14 PM             5,004 AnnotationCollectionI.pm
07/03/2007  07:14 PM             4,654 AnnotationI.pm
07/03/2007  07:14 PM    <DIR>          Assembly
07/03/2007  07:14 PM    <DIR>          Biblio
07/03/2007  07:14 PM            11,153 Biblio.pm
07/03/2007  07:14 PM    <DIR>          Cluster
07/03/2007  07:14 PM             4,100 ClusterI.pm
07/03/2007  07:14 PM    <DIR>          ClusterIO
07/03/2007  07:14 PM             7,792 ClusterIO.pm
07/03/2007  07:14 PM    <DIR>          Coordinate
07/03/2007  07:14 PM    <DIR>          Das
07/03/2007  07:14 PM            13,084 DasI.pm
07/03/2007  07:14 PM    <DIR>          DB
07/03/2007  07:14 PM             2,738 DBLinkContainerI.pm
07/03/2007  07:14 PM             2,882 DescribableI.pm
```

```
07/03/2007  07:14 PM    <DIR>          Event
07/03/2007  07:14 PM    <DIR>          Expression
07/03/2007  07:14 PM    <DIR>          Factory
07/03/2007  07:14 PM             5,327 FeatureHolderI.pm
07/03/2007  07:14 PM    <DIR>          Graphics
07/03/2007  07:14 PM             2,793 Graphics.pm
07/03/2007  07:14 PM             2,530 IdCollectionI.pm
07/03/2007  07:14 PM             5,146 IdentifiableI.pm
07/03/2007  07:14 PM    <DIR>          Index
07/03/2007  07:14 PM    <DIR>          LiveSeq
07/03/2007  07:14 PM            10,986 LocatableSeq.pm
07/03/2007  07:14 PM    <DIR>          Location
07/03/2007  07:14 PM            12,304 LocationI.pm
07/03/2007  07:14 PM    <DIR>          Map
07/03/2007  07:14 PM    <DIR>          MapIO
07/03/2007  07:14 PM             5,329 MapIO.pm
07/03/2007  07:14 PM    <DIR>          Matrix
07/03/2007  07:14 PM    <DIR>          Ontology
07/03/2007  07:14 PM    <DIR>          OntologyIO
07/03/2007  07:14 PM             8,235 OntologyIO.pm
07/03/2007  07:14 PM            17,252 Perl.pm
07/03/2007  07:14 PM    <DIR>          Phenotype
07/03/2007  07:14 PM            24,568 PrimarySeq.pm
07/03/2007  07:14 PM            20,679 PrimarySeqI.pm
07/03/2007  07:14 PM             7,472 Range.pm
07/03/2007  07:14 PM            11,972 RangeI.pm
07/03/2007  07:14 PM    <DIR>          Root
07/03/2007  07:14 PM    <DIR>          Search
07/03/2007  07:14 PM             5,435 SearchDist.pm
07/03/2007  07:14 PM    <DIR>          SearchIO
07/03/2007  07:14 PM            15,069 SearchIO.pm
07/03/2007  07:14 PM    <DIR>          Seq
07/03/2007  07:14 PM            37,739 Seq.pm
07/03/2007  07:14 PM             3,299 SeqAnalysisParserI.pm
07/03/2007  07:14 PM    <DIR>          SeqFeature
07/03/2007  07:14 PM            17,041 SeqFeatureI.pm
07/03/2007  07:14 PM             6,484 SeqI.pm
07/03/2007  07:14 PM    <DIR>          SeqIO
07/03/2007  07:14 PM            21,805 SeqIO.pm
07/03/2007  07:14 PM             9,799 SeqUtils.pm
07/03/2007  07:14 PM            50,828 SimpleAlign.pm
07/03/2007  07:14 PM             8,380 Species.pm
07/03/2007  07:14 PM    <DIR>          Structure
07/03/2007  07:14 PM    <DIR>          Symbol
07/03/2007  07:14 PM    <DIR>          Taxonomy
07/03/2007  07:14 PM             6,372 Taxonomy.pm
07/03/2007  07:14 PM    <DIR>          Tools
07/03/2007  07:14 PM    <DIR>          Tree
07/03/2007  07:14 PM    <DIR>          TreeIO
07/03/2007  07:14 PM             6,223 TreeIO.pm
07/03/2007  07:14 PM             3,251 UpdateableSeqI.pm
07/03/2007  07:14 PM    <DIR>          Variation
              39 File(s)        430,638 bytes
              38 Dir(s)  62,847,074,304 bytes free
```

示例 C.4

```
(Perl Script using bioperl modules from the bioperl Web site)
use Bio::Seq;
use Bio::SeqIO;

# create a sequence object of some DNA
my $seq = Bio::Seq->new(-id => 'testseq', -seq => 'CATGTAGATAG'
);

# print out some details about it
print "seq is ", $seq->length, " bases long\n";
print "reversed complement seq is ", $seq->revcom->seq, "\n";

# write it to a file in Fasta format
my $out = Bio::SeqIO->new(-file => '>testseq.fsa', -format => '
Fasta');
$out->write_seq($seq);

(Output)
seq is 11 bases long
reversed complement seq is CTATCTACATG

$ more testseq.fsa   <-- The FASTA file
>testseq
CATGTAGATAG
```

如需了解更多使用 BioPerl 的示例，请访问：

http://www.bioperl.org/wiki/Bptutorial#Quick_getting_started_scripts

BioPerl 带有一组产品级品质的脚本，它们位于 scripts/ 目录中。读者可以通过回答 make install 问题的方式来安装这些脚本。其默认目录路径是 /usr/bin。安装过程会把这些脚本复制到指定的目录中，然后将其 PLS 后缀更改为 pl，并在所有不带 bp_ 前缀的脚本名前面加上 bp_ 前缀。

C.5 资源

如需获取有关本章所述内容的更多信息，读者可以访问一些优秀的 Web 站点，亦可查阅某些优秀读物。作为一名初学者，读者可以尝试阅读下列书籍：

1. Walker, Sharon. *Biotechnology DeMystified*. McGraw-Hill, 2007.
2. Tisdall, James. *Beginning Perl for Bioinformatics*. O' Reilly & Associates, 2001.
3. Tisdall, James. *Mastering Perl for Bioinformatics*. O' Reilly & Associates, 2003.

附录 D　强悍与速度：CGI 和 mod_Perl

D.1　什么是 mod_Perl

有些人可能会说，CGI 已经是昨日黄花了，现在应当使用 PHP、ASP.NET 等。这是不对的。通过把 Perl 集成到 Apache 服务器中，mod_Perl 模块（原作者为 Doug MacEachern，遵循 Apache 软件证书）为用户提供了一条全新的创建动态内容的途径。mod_Perl 模块是 Apache 与 Perl 的联姻。它是一个开放源码的模块，为 Apache API 实现了一套 Perl 接口，从而能让已有 CGI 脚本运行得更快。

通过把 Perl 解释器嵌入到 Apache 服务器中，就没有必要在执行 CGI 程序时启动（fork）独立的外部解释器了。此外，所有模块与脚本将只编译一次，然后在剩下的服务器声明周期中，它们将始终位于缓冲区中，随时可以提供服务。因此，服务器只需把时间花费在执行那些已经载入并编译完毕的代码上，该过程将是非常快速的。

CGI.pm 的作者 Lincoln Stein 是这样描述 mod_Perl 的：

“mod_Perl 不仅仅是一种强大的 CGI 脚本。它为创建动态内容提供了一种全新的方式，因为它借助了 Apache Web 服务器的强大功能，譬如创建有状态的会话、自定义用户验证系统、智能代理服务器，等等。更奇妙的是，用户原有的 CGI 脚本仍然能够工作，而且速度还很快。”

目前，Apache 是 Internet 上最为流行的 Web 服务器，而 mod_Perl 则是最流行的扩展模块之一。借助 mod_Perl API，用户就能以 Perl 而不是 C 语言来编写定制的 Apache 模块，并能动态地对服务器进行配置，从而让基于复杂配置的服务器管理工作变得更为简便。用户还可编写认证和授权处理程序，以便允许或禁止使用某个页面、执行数据库查询操作、重写 HTTP 请求与 URL、以及允许或禁止日志请求，等等。

mod_Perl 模块还绑定了另外两个通用模块，能在不修改已有 Perl CGI 脚本的情况下直接运行它们（只要它们书写得较为规范，而不是“粗制滥造”）。这两个 Perl 模块都可以在 Perl 的 site/lib 库中找到；它们名叫 Apache::Registry.pm（或 ModPerl::Registry.pm）和 Apache::PerlRun.pm（或 ModPerl::PerlRun.pm）。这两个模块的区别在于，Registry.pm 会缓存所有的脚本，而 PerlRun.pm 不会这么做。缓冲（即编译脚本一次，然后存入内存）的优势在于其优秀的速度和持久性。如果脚本编写得较好，没有忘记初始化变量、关闭文件句柄、设置警告和调试开关，并基本遵循良好的 Perl 编程规范的话，上述机制才能正常工作。如果脚本编写得较为粗糙，则 PerlRun.pm 模块将是一个更安全的选择。每当收到一个调用脚本的请求时，一切都要从头开始，因为代码还都没有缓冲。此外

还有一个较大的优势，即 Perl 解释器只需载入一次，因此能够提升用户脚本的性能。mod_Perl 站点（http://Perl.apache.org/download/index.html）中介绍说：“标准的 Apache::Registry.pm 模块能为已有 CGI 脚本提供超过 100 倍的加速比，同时还能减小用户服务器的负载。”

有关使用 mod_Perl 和 Apache 的统计信息，请访问如图 D-1 所示站点。

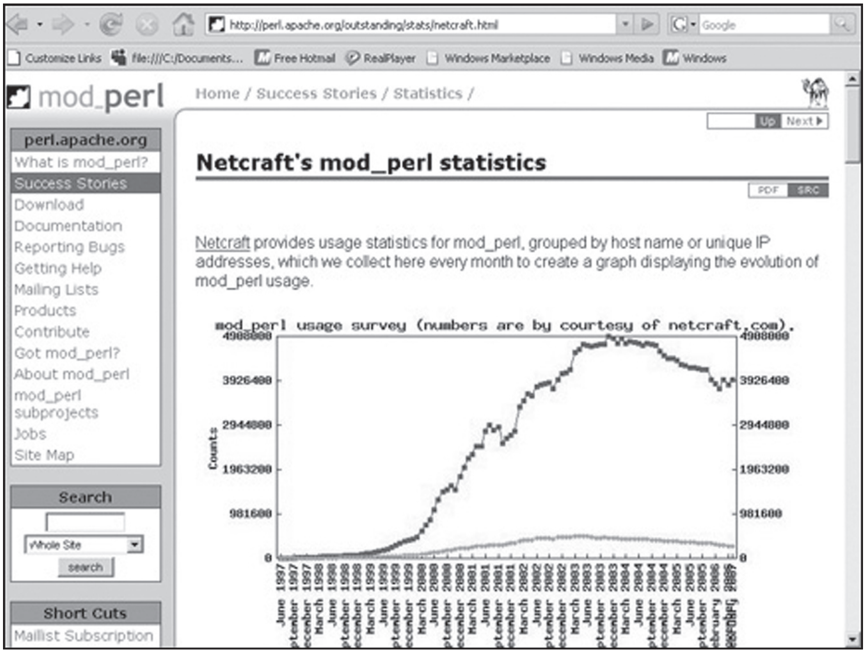


图 D-1

D.2 mod_Perl Web 站点

下面的内容在 mod_Perl 主页上都有讨论，该站点能为用户提供有关 mod_Perl 的所有知识，包括其安装、下载、文档、帮助、bug 等信息。这里显示了该站点的概要情况。

什么是 mod_Perl ?

mod_Perl 是 Apache 与 Perl 的联姻。

mod_Perl 把当前 Web 专业领域最强大也最成熟的两项技术联系到一起。

mp2 是针对 Apache HTTPD 服务器第 2.x.x 子版本提供的 mod_Perl。

mp1 是针对 Apache HTTPD 服务器第 1.3.x 子版本提供的 mod_Perl。

mp2 完全兼容于 HTTPD 2.0.x，并支持大部分 2.2.x 特性。

mod_Perl 未来的目标是继续支持 HTTPD——这项目标过去如此，未来也永远如此。

只需安装一份 mod_Perl，用户便能随心所欲地使用 Apache Web 服务器的强大能力。

成功案例

mod_Perl 是 Internet 上许多最繁忙也最先进的 Web 站点的幕后英雄。这里列举一些来自 mod_Perl 用户的成功案例；还有 mod_Perl 在全球范围内的使用情况。

下载

获取 mod_Perl 的源码或二进制发布包，并下载文档。

文档

mod_Perl 项目拥有大量文档，涉及 mod_Perl 1.0 与 2.0。用户若想了解有关 mod_Perl 的任何内容，均可在这里找到它。

Bug 报告

在解决某个 bug 之前，开发者必须首先重现它。用户最好能提供所有的相关信息，以帮助重现该 bug。但是，用户确实难以决定究竟该把哪些信息放到 bug 报告中。为了加速信息检索的过程，我们编写了一份指南文档，精确地解释了需要提供哪些信息。一般而言，bug 报告写得越好，bug 重现乃至修复就进行得越快。

获取帮助

解决用户碰到的 mod_Perl 问题: 借助 mod_Perl 邮件列表、mod_Perl 培训公司、或者借助某个商业化支持公司。不妨去找一个提供 mod_Perl 服务的 ISP。

邮件列表

mod_Perl 及其相关项目的邮件列表。

产品

已经有大量的软件能够与 mod_Perl 配合使用，并 / 或帮助用户编写自己的工程。

贡献

如何向 mod_Perl 社区做出自己的一份贡献。

了解 mod_Perl ?

有关 mod_Perl 的推介文档与资源。

有关 mod_Perl

有关 mod_Perl 历史的一般信息。

mod_Perl 子项目

在 mod_Perl 羽翼下维护的其他项目。

工作

找到一份梦寐以求的 mod_Perl 工作职位！

站点地图

从这张站点地图可以访问到本站点上的任意文档。

D.3　安装 mod_Perl

本节将描述安装 mod_Perl 以及为快速执行 CGI 脚本而配置 Apache 的一般步骤。读者应当已经安装并启动了最新版的 Perl 和 Apache，并且熟悉 CGI 的工作方式（详见第 16 章）。

下面这个示例演示了如何安装并在一个使用 Apache2.x 与 ActivePerl5.8.8 的窗口系统上运行 mod_Perl 脚本。

如要获取文档，请访问 mod_Perl 站点，http://Perl.apache.org/docs/index.html。如要获得 mod_Perl 源码或二进制发布包，请访问下载页面 http://Perl.apache.org/download/index.html。这里配置 Apache httpd.conf 文件的方式可能会略有不同，因为用到的 Perl 模块可能并不位于与之相同的库中。为了

搞清楚如何把本示例移植到 Apache 和 Perl 的当前版本，可能会花费一些时间。

mod_Perl 有两种不同的版本：

mp2——针对 Apache HTTPD 服务器第 2.x.x 子版本提供的 mod_Perl。

mp1——针对 Apache HTTPD 服务器第 1.3.x 子版本提供的 mod_Perl。

在本示例中，我们将使用 mp2，它完全兼容于 HTTPD 2.0.x，并支持大部分 2.2.x 特性。

为带有 PPM 的 ActiveState 安装 mod_Perl。 借助 ActiveState 的 PPM 工具，模块的安装过程变得越来越简便了。目前，该工具已经包含在每一种 ActivePerl 发布包中，不仅包括 Windows 版，还包括 UNIX 版本的 ActivePerl。该工具允许读者进行安装、移除、升级，并能管理 Perl 公共预编译 CPAN 模块的使用。

下面的示例在 Windows 系统上用到了 PPM 工具。

示例 D.1

```
C:\>ppm install mod_perl
Downloading ActiveState Package Repository packlist...done
Updating ActiveState Package Repository database...done
ppm install failed: Can't find any package that provide mod_perl

C:\>ppm install http://theoryx5.uwinnipeg
/ppms/mod_perl-2.0.ppd
Downloading mod_perl-2.0-2.0.3...done
Unpacking mod_perl-2.0-2.0.3...done
Generating HTML for mod_perl-2.0-2.0.3...done
Updating files in site area...done
Downloading mod_perl-2.0-2.0.3 install script...done
Running mod_perl-2.0-2.0.3 install script...
The Apache module mod_perl.so is needed to complete the installation,
and should be placed in your Apache2 modules directory. I will
now fetch and install this for you.

Fetching http://theoryx5.uwinnipeg.ca/ppms/x86/mod_perl-2.0.so ...
done!
Where should mod_perl.so be placed? [C:/Apache2/modules]
c:/wamp/Apache2/modu

mod_perl.so has been successfully installed to c:/wamp/Apache2/modules
done
 465 files installed
```

Apache 的 conf 目录（配置文件）

示例 D.2

```
cd c:\wamp\Apache2\conf
C:\wamp\Apache2\conf>dir /b
alias
httpd.conf
httpd.default.conf
magic
magic.default
mime.types
mime.types.default
```

编辑 Apache 的 httpd.conf 文件

接着必须修改 Apache 的 httpd.conf 文件，添加如下各行：

示例 D.3

```
#LoadModule mime_magic_module modules/mod_mime_magic.so
#LoadModule proxy_module modules/mod_proxy.so
#LoadModule proxy_connect_module modules/mod_proxy_connect.so
#LoadModule proxy_http_module modules/mod_proxy_http.so
#LoadModule proxy_ftp_module modules/mod_proxy_ftp.so
LoadModule negotiation_module modules/mod_negotiation.so
#LoadModule rewrite_module modules/mod_rewrite.so
LoadModule setenvif_module modules/mod_setenvif.so
#LoadModule speling_module modules/mod_speling.so
#LoadModule status_module modules/mod_status.so
#LoadModule unique_id_module modules/mod_unique_id.so
LoadModule userdir_module modules/mod_userdir.so
#LoadModule usertrack_module modules/mod_usertrack.so
#LoadModule vhost_alias_module modules/mod_vhost_alias.so
LoadModule php5_module "c:/wamp/php/php5apache2.dll"
LoadModule perl_module modules/mod_perl.so
#
# ExtendedStatus controls whether Apache will generate "full" status
# information (ExtendedStatus On) or just basic information (ExtendedStatus
# Off) when the "server-status" handler is called. The default
is Off.
===================================
```

为 mod_Perl 脚本创建位置。 现在我们将创建存放 mod_Perl 程序的目录 / 文件夹。这里不会把 CGI 脚本存入同样的 cgi-bin 目录，而是将它们移入这个新目录中，该目录名叫 c:/home/stas/modperl。

在本示例中，我们将把 Perl 脚本保存到：

c:/home/stas/modperl

Apache Perl 模块保存在何处？

根据所用 Perl 版本的不同，负责运行已有 Perl CGI 脚本的 Apache 模块将位于标准库中，其路径可从 @INC 数组中取得。

示例 D.4

```
cd /c/ActivePerl/site/lib/ModPerl
$ ls
BuildMM.pm          Manifest.pm          RegistryLoader.pm
BuildOptions.pm     MapUtil.pm           RegistryPrefork.pm
CScan.pm            MethodLookup.pm      StructureMap.pm
Code.pm             ParseSource.pm       TestReport.pm
Config.pm          PerlRun.pm          TestRun.pm
Const.pm            PerlRunPrefork.pm    TypeMap.pm
FunctionMap.pm      Registry.pm          Util.pm
Global.pm           RegistryBB.pm        WrapXS.pm
MM.pm               RegistryCooker.pm
```

PerlRequire 指令

然后，我们将在 Apache 的 conf 目录中为最常用的 Perl 模块创建一个文件。该文件叫作 extras.pl，含有 mod_Perl 最常用到的模块。通过把该文件名赋值给 Apache 配置文件中的 PerlRequire 指令，便能在每次启动 Apache 服务器时载入这些模块。（请注意，大多数有关添加模块的文档中都把 Registry.pm 模块称为 Apache::Registry。但在当前最新版本的 ActiveState Perl 中，该模块的名字已经成了 ModPerl::Registry。）

示例 D.5

```
C:\wamp\Apache2\conf>more extras.pl
use ModPerl::Util ();
  use Apache2::RequestRec ();
  use Apache2::RequestIO ();
  use Apache2::RequestUtil ();
  use Apache2::ServerRec ();
  use Apache2::ServerUtil ();
  use Apache2::Connection ();
  use Apache2::Log ();
  use Apache2::Const -compile => ':common';
  use APR::Const -compile => ':common';
  use APR::Table ();
  use Apache2::compat ();
  use ModPerl::Registry ();
  use CGI ();
  1;
```

编辑 Apache 的 httpd.conf 文件

现在我们将编辑 httpd.conf 文件，它位于 Apache 目录中。对于这个 Windows 系统上的 Apache 而言，conf 目录 / 文件夹位于 Apache2 目录下面。

Apache 的 conf 目录

示例 D.6

```
C:\wamp\Apache2>dir /b
ABOUT_APACHE.txt
bin
cgi-bin
CHANGES.txt
conf
error
icons
include
INSTALL.txt
lib
LICENSE.txt
logs
modules
NOTICE.txt
README.txt
```

我们现在会 cd 到 conf 目录中，并找到 httpd.conf 文件。

Apache 的配置文件 httpd.conf

示例 D.7

```
C:\wamp\Apache2>cd conf

C:\wamp\Apache2\conf>dir /b
alias
extra.pl
httpd.conf
httpd.default.conf
magic
magic.default
mime.types
mime.types.default
```

然后，我们会编辑 httpd.conf 文件，并将示例 D.8 中的各行添加到该文件的底部。（这里每一行前面的号码仅仅是为了便于在示例后面的解释部分中解释每一行内容。）

编辑 Apache 的配置文件

示例 D.8

```
1  Alias /perl/ c:/home/stas/modperl/

2  LoadFile "c:/ActivePerl/bin/perl58.dll"

3  PerlRequire "c:/wamp/Apache2/conf/extras.pl"

4  PerlModule ModPerl::Registry

5  <Location /perl/>
6      SetHandler perl-script
7      PerlHandler ModPerl::Registry
8      Options +ExecCGI
9      PerlSendHeader On
10   Allow from All
    </Location>
```

解释

1. 本行指定了 mod_Perl 脚本所在位置的别名。/Perl/ 是该别名的名称，c:/home/stas/modperl 则是其实际位置。如果服务器是本地的话，用户可以在浏览器窗口的位置框中输入 http://localhost/per;/mod_script_filename。
2. LoadFile 指示符能够保证安装正确版本的 Perl 解释器。
3. extras.pl 文件含有常用的公共 mod_Perl 模块。PerlRequire 则会让 Apache 在每次启动服务器时都载入这些文件。
4. PerlModule 指示符用于载入 Registry.pm 模块。（有关 mod_Perl 的许多文档都会让用户使用 Apache::Registry。而这里正在使用的则是 ModPerl::Registry。请检查 @INC 数组，并查看安装有 Registry 模块的库。当使用该指示符或者其他任何需要用到 Registry 模块的指示符时，

请确保模块名称的正确性。如果不想缓存脚本，只需再找到 Registry 的位置将 Registry 更改为 PerlRun 便可。）

5. Location 部分定义了一些在 Apache 处理 URI 请求时应当遵循的规则，这些 URI 都列举在我们之前创建的名叫 /Perl/ 的别名中。
   我们将用于执行 CGI 脚本的模块叫作 Registry.pm。而在这个 ActivePerl 安装版中，用户可在数组 @INC 列出的站点库中找到该模块。
6. 在生成内容时，该指示符将赋予 mod_Perl 模块，以便对该模块进行处理。
7. 这个指示符告诉 Apache 应当使用什么模块去生成内容。模块决定了在每次处理请求时将如何载入这些内容。
8. 该选项告诉 Apache 该 CGI 脚本是可以接受的，并且它们也拥有执行权限。
9. 当该指示符处于打开状态时，mod_Perl 将会创建格式正确的 HTTP 首部。
10. 当设置了 Allow from All 时，任何客户端都将有权执行该 IP 地址和域上的脚本。

mod_Perl 安装了吗？

检查 mod_Perl 是否功能正常的方法有很多种。读者可以在重启服务器后找到 Apache 的错误日志，如下所示：

```
cd c:\wamp\logs
C:\wamp\logs>dir /b
access.log
apache_error.log
log_dir
mysql_error.log
php_error.log
```

在重启服务器后检查 Apache 错误日志。这时应当能看到如下所示的部分内容：

```
    [Mon Jul 02 14:08:11 2007] [notice]
Apache/2.0.55 (Win32) PHP/5.1.4
    mod_perl/2.
    0.3 Perl/v5.8.8 configured -- resuming
normal operations
    [Mon Jul 02 14:08:11 2007] [notice] Server built: Oct  9 2005 19:16:56
    [Mon Jul 02 14:08:11 2007] [notice] Parent: Created child process 5008
```

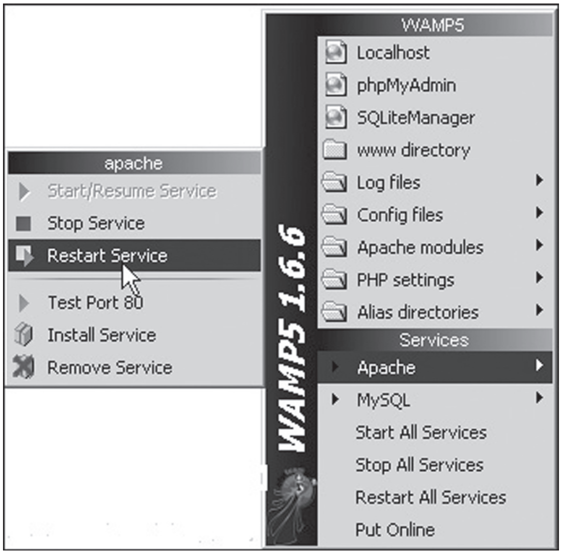


图 D-2 使用 WAMP5 管理 Apache

借助 CGI 脚本检查 mod_Perl

在 Apache 根目录下的传统 cgi-bin 目录中，读者可以找到一个标准的测试文件，该文件是随 Apache 发布的。该文件叫作 printenv.pl。请把该文件拷贝到刚才创建的新 mod_Perl 目录中，在本例中就是 c:/home/stas/modperl，该路径在 http.conf 文件中的别名是 /Perl/。请确保其权限得到了正确的设置。该服务器应当为脚本提供执行权限。如果没有的话，则在浏览器中会出现如图 D-3 所示的错误：

现在，请访问浏览器中的 URL，并像图 D-4 那样执行 printenv.pl 文件。该文件能显示所有的环境变量，包括 MOD_PERL 和 MOD_PERL_API_VERSION 变量。



图 D-3　来自 Apache 的权限错误

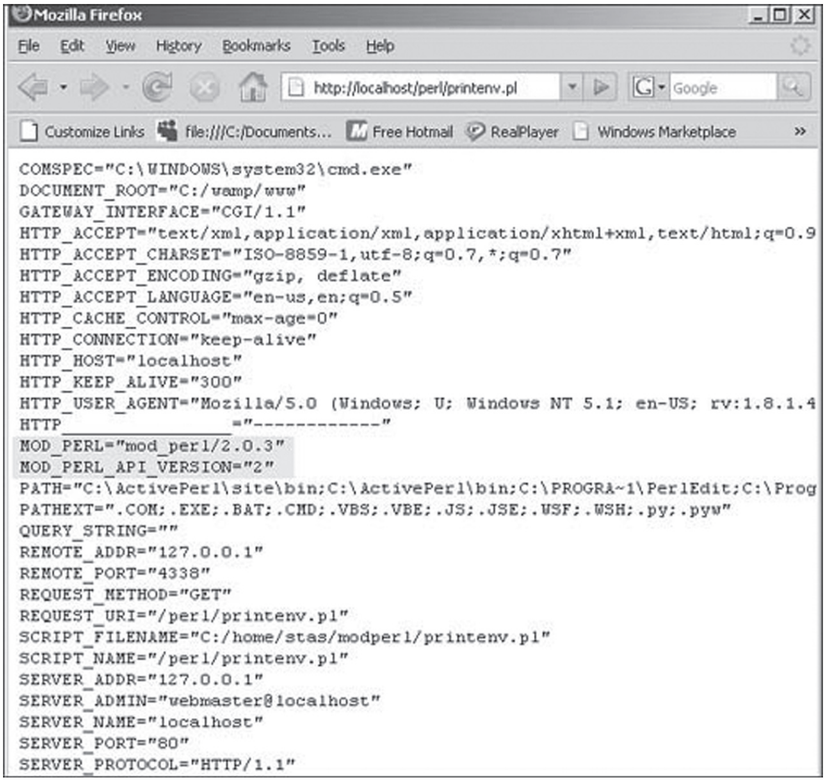


图 D-4　执行 Perl 脚本 printenv.pl

尝试另一个 CGI 脚本：

请把下列 CGI 脚本放入 mod_Perl 目录中。该脚本用到了 CGI.pm，其样式是面向对象的。在安装好 mod_Perl 之后，CGI Perl 脚本都应当从该目录中执行。其运行情况如图 D-5 和 D-6 所示。

示例 D.9

```
#!c:/ActivePerl/bin/perl.exe
use strict;
use CGI;
my $q = CGI->new;
print $q->header, $q->start_html(-title=>'First mod_perl Try',
                                  -BGCOLOR=>'lightgreen'),
           $q->h1('Hi, mod_perl is working!!'),
           $q->end_html;
```

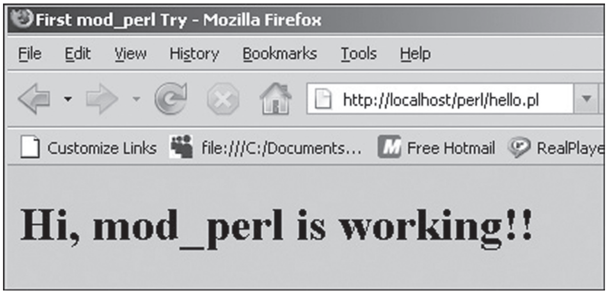


图 D-5　借助 mod_Perl 运行 CGI 脚本

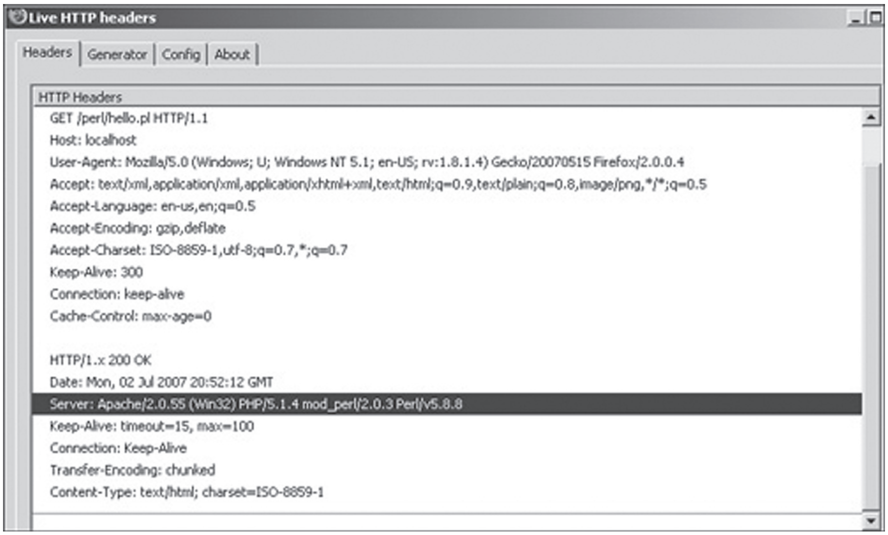


图 D-6　Firefox 插件 Live HTTP headers 能显示 HTTP 请求 / 响应

UNIX/Linux 安装

下面这段脚本来自于：

http://www.Perl.com/pub/a/2002/03/22/modperl.html。

它能检查用户正在使用的 Linux/UNIX 系统版本。

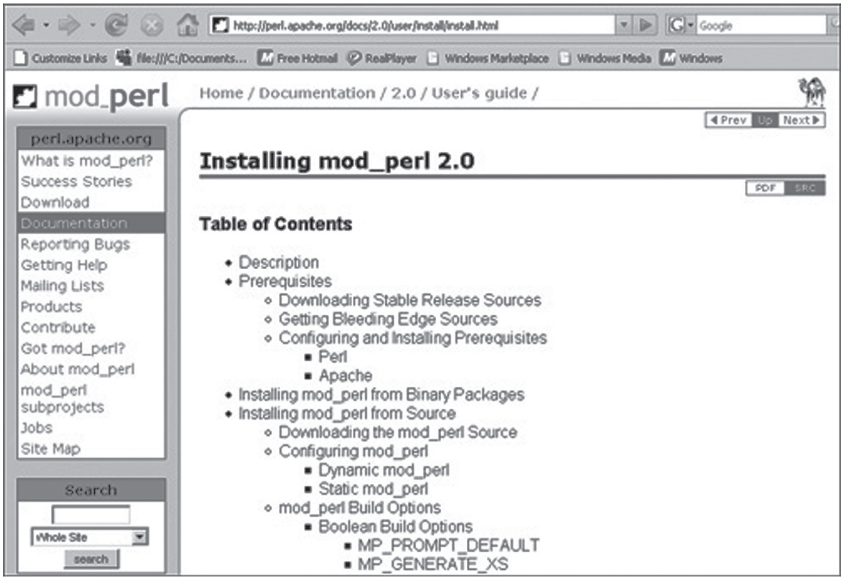


图 D-7 在 Apache 官方站点咨询下载事宜

示例 D.10

```
   % cd  /usr/src
%lwp-download http://www.apache.org/dist/httpd/binaries/httpd-2.0.55-sparc-
sun-solaris2.8.tar.gz
  % lwp-download http://perl.apache.org/dist/mod_perl-2.0-current.tar.gz
  % tar -zvxf apache_1.3.20.tar.gz
  % tar -zvxf mod_perl-1.26.tar.gz
  % cd mod_perl-1.26
  % perl Makefile.PL APACHE_SRC=../apache_1.3.20/src \
    DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
  % make && make test && make install
  % cd ../apache_2.0.55  (Check for the correct number here)
  % make install
```

mod_Perl 文档。mod_Perl 项目拥有大量文档，涉及 mod_Perl 1.0 与 2.0。用户若想了解有关 mod_Perl 的任何内容，均可在这里找到它。

mod_Perl 文档中用到的规范。

在文档中用到了很多易于理解的规范；如果读者仍然怀有疑惑，请在这里查看帮助信息。

mod_Perl 1.0 文档

与 mod_Perl 1,0 版相关的一组文档。

mod_Perl 2.0 文档

与 mod_Perl 2,0 版相关的一组文档。

通用文档

读者在这里可以找到有关 mod_Perl 的通用文档，以及其他一些与 mod_Perl 不直接相关，但却有益于使用 mod_Perl 的信息。这里的大部分文档都适用于 mod_Perl 1.0 和 2.0。

离线资源

mod_Perl 相关书籍、文档、演示，以及指向其他相关主题站点的链接。

D.4 资源

1. Bekman, Stas and Cholet, Eric. *Practical mod_Perl*. O'Reilly and Associates, Inc.

2. Stein, Lincoln and MacEastern, Doug. *Writing Apache Modules with Perl and C*. O'Reilly and Associates, Inc.